Software Testing

Principles and Practices

Srinivasan Desikan
Director of Quality Assurance,
Agile Software Enterprise, Pvt. Ltd.
Bangalore, India

Gopalaswamy Ramesh
Consultant and Adjunct Professor,
International Institute of Information Technology,
Bangalore, India

PEARSON
Education

Contents

Part III Select Topics in Specialized Testing

Part IV People and Organizational Issues in Testing

Part V Test Management and Automation

Principles of Testing

CHAPTER 1

In this chapter—

- ✓ Context of testing in producing software
- ✓ About this chapter
- ✓ The incomplete car
- ✓ Dijkstra's doctrine
- ✓ A test in time!
- ✓ The cat and the saint
- ✓ Test the tests first!
- ✓ The pesticide paradox
- ✓ The convoy and the rags
- ✓ The policemen on the bridge
- ✓ The ends of the pendulum
- ✓ Men in black
- ✓ Automation syndrome
- ✓ Putting it all together

1.1 CONTEXT OF TESTING IN PRODUCING SOFTWARE

Almost everything we use today has an element of software in it. In the early days of evolution of software, the users of software formed a small number compared to the total strength of an organization. Today, in a typical workplace (and at home), just about everyone uses a computer and software. Administrative staff use office productivity software (replacing the typewriters of yesteryears). Accountants and finance people use spreadsheets and other financial packages to help them do much faster what they used to do with calculators (or even manually). Everyone in an organization and at home uses e-mail and the Internet for entertainment, education, communication, interaction, and for getting any information they want. In addition, of course, the "technical" people use programming languages, modeling tools, simulation tools, and database management systems for tasks that they were mostly executing manually a few years earlier.

In god we trust, everything else we test

The above examples are just some instances where the use of software is "obvious" to the users. However, software is more ubiquitous and pervasive than seen in these examples. Software today is as common as electricity was in the early part of the last century. Almost every gadget and device we have at home and at work is embedded with a significant amount of software. Mobile phones, televisions, wrist watches, and refrigerators or any kitchen equipment all have embedded software.

Another interesting dimension is that software is being used now in mission critical situations where failure is simply unacceptable. There is no way one can suggest a solution of "please shutdown and reboot the system" for a software that is in someone's pacemaker! Almost every service we have taken for granted has software. Banks, air traffic controls, cars are all powered by software that simply cannot afford to fail. These systems have to run reliably, predictably, all the time, every time.

This pervasiveness, ubiquity, and mission criticality places certain demands on the way the software is developed and deployed.

First, an organization that develops any form of software product or service must put in every effort to drastically reduce and, preferably, eliminate any defects in each delivered product or service. Users are increasingly intolerant of the hit-and-miss approach that characterized software products. From the point of view of a software development organization also, it may not be economically viable to deliver products with defects. For instance, imagine finding a defect in the software embedded in a television after it is shipped to thousands of customers. How is it possible to send "patches" to these customers and ask them to "install the patch?" Thus, the only solution is to do it right the first time, before sending a product to the customer.

Second, defects are unlikely to remain latent for long. When the number of users was limited and the way they used the product was also predictable

(and highly restricted), it was quite possible that there could be defects in the software product that would never get detected or uncovered for a very long time. However, with the number of users increasing, the chances of a defect going undetected are becoming increasingly slim. If a defect is present in the product, *someone* will hit upon it sooner than later.

Third, the nature of usage of a product or a service is becoming increasingly unpredictable. When bespoke software is developed for a specific function for a specific organization (for example, a payroll package), the nature of usage of the product can be predictable. For example, users can only exercise the specific functionality provided in the bespoke software. In addition, the developers of the software know the users, their business functions, and the user operations. On the other hand, consider a generic application hosted on the Internet. The developers of the application have no control over how someone will use the application. They may exercise untested functionality; they may have improper hardware or software environments; or they may not be fully trained on the application and thus simply use the product in an incorrect or unintended manner. Despite all this "mishandling," the product should work correctly.

Finally, the consequence and impact of every single defect needs analysis, especially for mission critical applications. It may be acceptable to say that 99.9% of defects are fixed in a product for a release, and only 0.1% defects are outstanding. It appears to be an excellent statistics to go ahead and release the product. However, if we map the 0.1% failure in mission critical applications, the data will look like this.

- A total of 10,000 incorrect surgery operations per week.
- Three airplane crashes every day.
- No electricity for five hours every week.

For sure, the above data is unacceptable for any individual, organization, or government. Providing a workaround, such as "In case of fire, wear this dress," or documenting a failure, such as "You may lose only body parts in case of a wrong airplane landing" would not be acceptable in cases of mission critical applications.

This book focuses on software testing. Traditionally, testing is defined as being narrowly confined to testing the program code. We would like to consider testing in a broader context as encompassing all activities that address the implications of producing quality products discussed above. Producing a software product entails several phases (such as requirements gathering, design, and coding) in addition to testing (in the traditional sense of the term). While testing is definitely one of the factors (and one of the phases) that contributes to a high quality product, it alone cannot *add* quality to a product. Proper interaction of testing with other phases is essential for a good product. These interactions and their impact are captured in the grid in Figure 1.1.

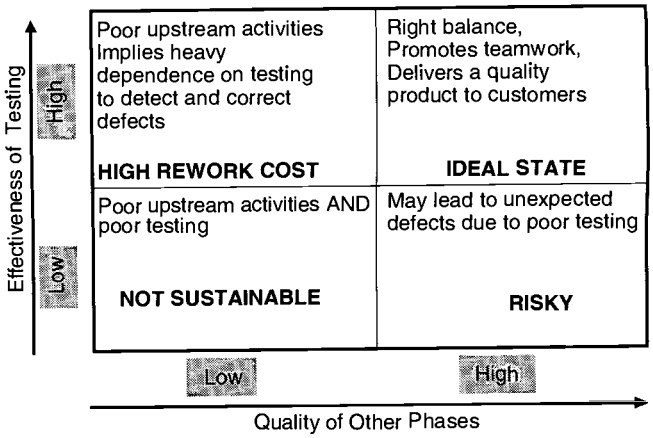


Figure 1.1
Relationship of effectiveness of testing to quality of other phases.

If the quality of the other phases is low and the effectiveness of testing is low (lower left-hand corner of the grid), the situation is not sustainable. The product will most likely go out of business very soon. Trying to compensate for poor quality in other phases with increased emphasis on the testing phase (upper left-hand corner of the grid) is likely to put high pressure on everyone as the defects get detected closer to the time the product is about to be released. Similarly, blindly believing other phases to be of high quality and having a poor testing phase (lower right-hand side of the grid) will lead to the risky situation of unforeseen defects being detected at the last minute. The ideal state of course is when high quality is present in all the phases including testing (upper right-hand corner of the grid). In this state, the customers feel the benefits of quality and this promotes better teamwork and success in an organization.

1.2 ABOUT THIS CHAPTER

In this chapter, we discuss some of the basic principles of testing. We believe that these principles are fundamental to the objective of testing, namely, to provide quality products to customers. These principles also form the motivation for the rest of the book. Thus this chapter acts as an anchor for the rest of the book.

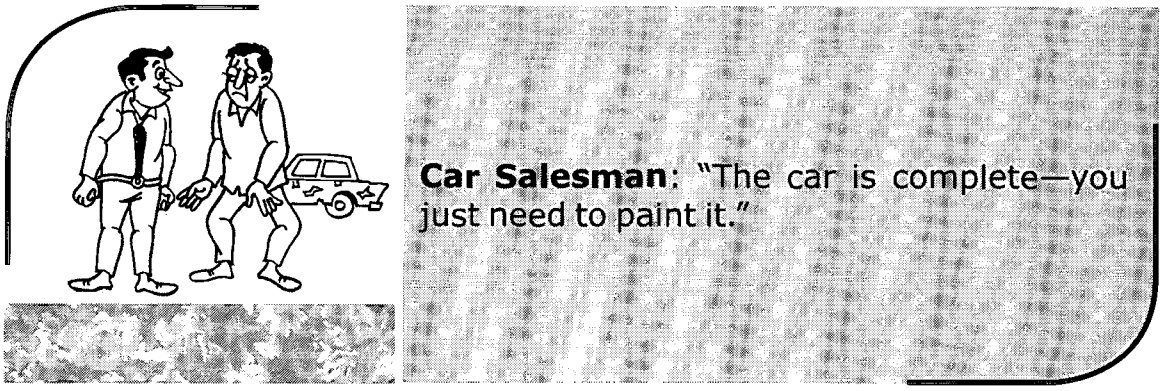
The fundamental principles of testing are as follows.

1. The goal of testing is to find defects before customers find them out.
2. Exhaustive testing is not possible; program testing can only show the presence of defects, never their absence.
3. Testing applies all through the software life cycle and is not an end-of-cycle activity.
4. Understand the reason behind the test.

5. Test the tests first.
6. Tests develop immunity and have to be revised constantly.
7. Defects occur in convoys or clusters, and testing should focus on these convoys.
8. Testing encompasses defect prevention.
9. Testing is a fine balance of defect prevention and defect detection.
10. Intelligent and well-planned automation is key to realizing the benefits of testing.
11. Testing requires talented, committed people who believe in themselves and work in teams.

We will take up each of these principles in the subsequent sections. Where appropriate, we will illustrate the principle with a simple story from outside the arena of information technology to drive home the point.

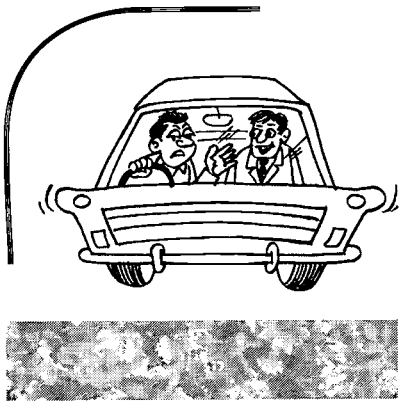
1.3 THE INCOMPLETE CAR



Eventually, whatever a software organization develops should meet the needs of the customer. Everything else is secondary. Testing is a means of making sure that the product meets the needs of the customer.

We would like to assign a broader meaning to the term "customer." It does not mean just *external* customers. There are also *internal* customers. For example, if a product is built using different components from different groups within an organization, the users of these different components should be considered customers, even if they are from the same organization. Having this customer perspective enhances the quality of all the activities including testing.

We can take the internal customer concept a step further where the development team considers the testing team as its internal customer. This way we can ensure that the product is built not only for usage requirements



Sales representative / Engineer: "This car has the best possible transmission and brake, and accelerates from 0 to 80 mph in under 20 seconds!"
Customer: "Well, that may be true, but unfortunately it accelerates (even faster) when I press the brake pedal!"

but also for testing requirements. This concept improves "testability" of the product and improves interactions between the development and testing teams.

We would like to urge the reader to retain these two perspectives—customer perspective and perspective of quality not being an add-on in the end, but built in every activity and component right from the beginning—throughout the book.

Testing should focus on finding defects before customers find them.

If our job is to give a complete car to the customer (and not ask the customers to paint the car) and if our intent is to make sure the car works as expected, without any (major) problems, then we should ensure that we catch and correct all the defects in the car ourselves. This is the fundamental objective of testing. Anything we do in testing, it behoves us to remember that.

1.4 DIJKSTRA'S DOCTRINE

Consider a program that is supposed to accept a six-character code and ensure that the first character is numeric and rests of the characters are alphanumeric. How many combinations of input data should we test, if our goal is to test the program *exhaustively*?

The first character can be filled up in one of 10 ways (the digits 0–9). The second through sixth characters can each be filled up in 62 ways (digits 0–9, lower case letters a–z and capital letters A–Z). This means that we have a total of $10 \times (62^5)$ or 9,161,328,320 valid combinations of values to test. Assuming that each combination takes 10 seconds to test, testing all these valid combinations will take approximately 2,905 years!

Therefore, after 2,905 years, we may conclude that all valid inputs are accepted. But that is not the end of the story—what will happen to the program when we give *invalid* data? Continuing the above example, if we assume there are 10 punctuation characters, then we will have to spend

a total of 44,176 years to test all the valid and invalid combinations of input data.

All this just to accept one field and test it exhaustively. Obviously, exhaustive testing of a real life program is *never* possible.

All the above mean that we can choose to execute only a subset of the tests. To be effective, we should choose a subset of tests that can uncover the maximum number of errors. We will discuss in Chapter 4, on Black Box Testing, and Chapter 3, on White Box Testing, some techniques such as equivalence partitioning, boundary value analysis, code path analysis, and so on which help in identifying subsets of test cases that have a higher likelihood of uncovering defects.

Testing can only prove the presence of defects, never their absence.

Nevertheless, regardless of which subset of test cases we choose, we can never be 100% sure that there are no defects left out. But then, to extend an old cliché, nothing can be certain other than death and taxes, yet we live and do other things by judiciously managing the uncertainties.

1.5 A TEST IN TIME!

Defects in a product can come from any phase. There could have been errors while gathering initial requirements. If a wrong or incomplete requirement forms the basis for the design and development of a product, then that functionality can never be realized correctly in the eventual product. Similarly, when a product design—which forms the basis for the product development (*a la* coding)—is faulty, then the code that realizes the faulty design will also not meet the requirements. Thus, an essential condition should be that every phase of software development (requirements, design, coding, and so on) should catch and correct defects at that phase, without letting the defects seep to the next stage.

Let us look at the cost implications of letting defects seep through. If, during requirements capture, some requirements are erroneously captured and the error is not detected until the product is delivered to the customer, the organization incurs extra expenses for

- performing a wrong design based on the wrong requirements;
- transforming the wrong design into wrong code during the coding phase;
- testing to make sure the product complies with the (wrong) requirement; and
- releasing the product with the wrong functionality.

In Figure 1.2 the defects in requirements are shown in gray. The coloured figure is available on page 457. As you can see, these gray boxes are carried forward through three of the subsequent stages—design, coding, and testing.

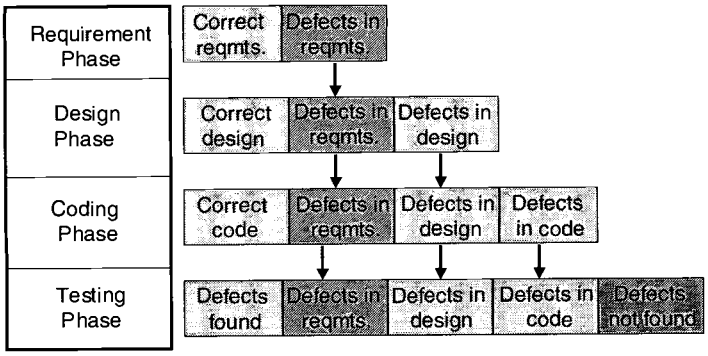


Figure 1.2
How defects from early phases add to the costs.

The cost of building a product and the number of defects in it increase steeply with the number of defects allowed to seep into the later phases.

When this erroneous product reaches the customer after the testing phase, the customer may incur a potential downtime that can result in loss of productivity or business. This in turn would reflect as a loss of goodwill to the software product organization. On top of this loss of goodwill, the software product organization would have to redo all the steps listed above, in order to rectify the problem.

Similarly, when a defect is encountered during the design phase (though the requirements were captured correctly, depicted by yellow), the costs of all of the subsequent phases (coding, testing, and so on) have to be incurred multiple times. However, presumably, the costs would be lower than in the first case, where even the requirements were not captured properly. This is because the design errors (represented by yellow boxes) are carried forward only to the coding and testing phases. Similarly, a defect in the coding phase is carried forward to the testing phase (green boxes). Again, as fewer phases are affected by this defect (compared to requirements defects or design defects), we can expect that the cost of defects in coding should be less than the earlier defects. As can be inferred from the above discussion, the cost of a defect is compounded depending on the delay in detecting the defect.

Hence, smaller the lag time between defect injection (i.e., when the defect was introduced) and defect detection (i.e., when the defect was encountered and corrected), lesser are the unnecessary costs. Thus, it becomes essential

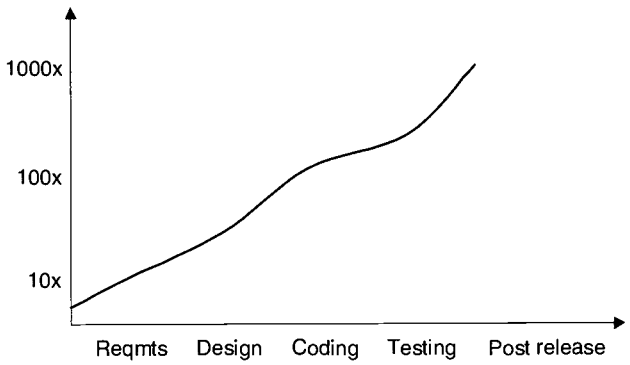


Figure 1.3
Compounding effect of defects on software costs.

to catch the defects as early as possible. Industry data has reaffirmed these findings. While there is no consensus about the costs incurred due to delay in defect detection, a defect introduced during the requirement phase that makes it to the final release may cost as much as a thousand times the cost of detecting and correcting the defect during requirements gathering itself.

1.6 THE CAT AND THE SAINT

A saint sat meditating. A cat that was prowling around was disturbing his concentration. Hence he asked his disciples to tie the cat to a pillar while he meditated. This sequence of events became a daily routine. The tradition continued over the years with the saint's descendents and the cat's descendents. One day, there were no cats in the hermitage. The disciples got panicky and searched for a cat, saying, *"We need a cat. Only when we get a cat, can we tie it to a pillar and only after that can the saint start meditating!"*

Testing requires asking about and understanding what you are trying to test, knowing what the correct outcome is, and why you are performing any test. If we carry out tests without understanding why we are running them, we will end up in running inappropriate tests that do not address what the product should do. In fact, it may even turn out that the product is modified to make sure the tests are run successfully, even if the product does not meet the intended customer needs!

"Why one tests" is as important as "What to test" and "How to test."

Understanding the rationale of why we are testing certain functionality leads to different types of tests, which we will cover in Part II of the book. We do white box testing to check the various paths in the code and make sure they are exercised correctly. Knowing which code paths should be exercised for a given test enables making necessary changes to ensure that appropriate paths are covered. Knowing the external functionality of what the product should do, we design black box tests. Integration tests are used to make sure that the different components fit together. Internationalization testing is used to ensure that the product works with multiple languages found in different parts of the world. Regression testing is done to ensure that changes work as designed and do not have any unintended side-effects.

1.7 TEST THE TESTS FIRST!

An audiologist was testing a patient, telling her, "I want to test the range within which you can hear. I will ask you from various distances to tell me your name, and you should tell me your name. Please turn back and answer." The patient understood what needs to be done.

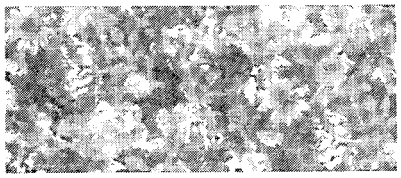
Doctor (from 30 feet): What is your name?

...

Doctor (from 20 feet): What is your name?

...

Doctor (from 10 feet): What is your name?
Patient: For the third time, let me repeat, my name is Sheela!



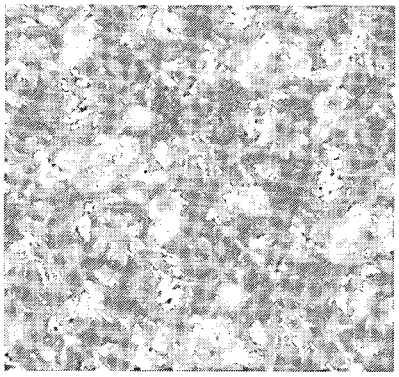
From the above example, it is clear that it is the audiologist who has a hearing problem, not the patient! Imagine if the doctor prescribed a treatment for the patient assuming that the latter could not hear at 20 feet and 30 feet.

Tests are also artifacts produced by human beings, much as programs and documents are. We cannot assume that the tests will be perfect either! It is important to make sure that the tests themselves are not faulty before we start using them. One way of making sure that the tests are tested is to document the inputs and expected outputs for a given test and have this description validated by an expert or get it counter-checked by some means outside the tests themselves. For example, by giving a known input value and separately tracing out the path to be followed by the program or the process, one can manually ascertain the output that should be obtained. By comparing this "known correct result" with the result produced by the product, the confidence level of the test and the product can be increased. The practices of reviews and inspection and meticulous test planning discussed in Chapter 3 and Chapter 15 provide means to test the test.

Test the tests first—a defective test is more dangerous than a defective product!

1.8 THE PESTICIDE PARADOX

Defects are like pests; testing is like designing the right pesticides to catch and kill the pests; and the test cases that are written are like pesticides. Just like pests, defects develop immunity against test cases! As and when we write new test cases and uncover new defects in the product, other defects that were "hiding" underneath show up.



Every year, pests of various types attack fields and crops. Agriculture and crop experts find the right antidote to counter these pests and design pesticides with new and improved formulae. Interestingly, the pests get used to the new pesticides, develop immunity, and render the new pesticides ineffective. In subsequent years, the old pesticides have to be used to kill the pests which have not yet developed this immunity and new and improved formulae that can combat these tougher variants of pests have to be introduced. This combination of new and old pesticides could sometimes even hinder the effectiveness of the (working) old pesticide. Over time, the old pesticides become useless. Thus, there is a constant battle between pests and pesticides to get ahead of the other. Sometimes pesticides win, but in a number of cases, the pests do succeed to defy the latest pesticides. This battle results in a constant churning and evolution of the nature and composition of pesticides.

Tests are like pesticides—you have to constantly revise their composition to tackle new pests (defects).

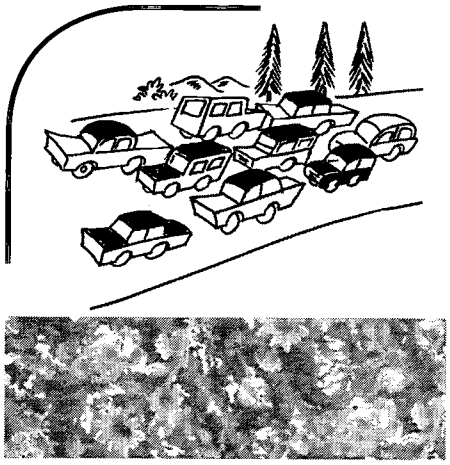
There are two possible ways to explain how products develop this "immunity" against test cases. One explanation is that the initial tests go a certain distance into the code and are stopped from proceeding further because of the defects they encounter. Once these defects are fixed, the tests proceed further, encounter newer parts of the code that have not been dealt with before, and uncover new defects. This takes a "white box" or a code approach to explain why new defects get unearthed with newer tests.

A second explanation for immunity is that when users (testers) start using (exercising) a product, the initial defects prevent them from using the full external functionality. As tests are run, defects are uncovered, and problems are fixed, users get to explore new functionality that has not been used before and this causes newer defects to be exposed. This "black box" view takes a functionality approach to explain the cause for this "more we test more defects come up" phenomenon.

An alternative way of looking at this problem is not that the defects develop immunity but the tests go deeper to further diagnose a problem and thus eventually "kill the defects." Unfortunately, given the complex nature of software and the interactions among multiple components, this final kill happens very rarely. Defects still survive the tests, haunt the customers, and cause untold havoc.

The need for constantly revising the tests to be run, with the intent of identifying new strains of the defects, will take us to test planning and different types of tests, especially regression tests. Regression tests acknowledge that new fixes (pesticides) can cause new "side-effects" (new strains of pests) and can also cause some older defects to appear. The challenge in designing and running regression tests centers around designing the right tests to combat new defects introduced by the immunity acquired by a program against old test cases. We will discuss regression tests in Chapter 8.

1.9 THE CONVOY AND THE RAGS



All of us experience traffic congestions. Typically, during these congestions, we will see a convoy effect. There will be stretches of roads with very heavy congestions, with vehicles looking like they are going in a convoy. This will be followed by a stretch of smooth sailing (rather, driving) until we encounter the next convoy.

Testing can only find a part of defects that exist in a cluster; fixing a defect may introduce another defect to the cluster.

Defects in a program also typically display this convoy phenomenon. They occur in clusters. Glenford Myers, in his seminal work on software testing [MYER-79], proposed that the *probability of the existence of more errors in a section of a program is proportional to the number of errors already found in that section.*

This may sound counter-intuitive, but can be logically reasoned out. A fix for one defect generally introduces some instability and necessitates another fix. All these fixes produce side-effects that eventually cause the convoy of defects in certain parts of the product.

From a test planning perspective, this means that if we find defects in a particular part of product, more—not less—effort should be spent on testing that part. This will increase the return on investments in testing as the purpose of testing is find the defects. This also means that whenever a product undergoes any change, these error-prone areas need to be tested as they may get affected. We will cover these aspects in Chapter 8, Regression Testing.

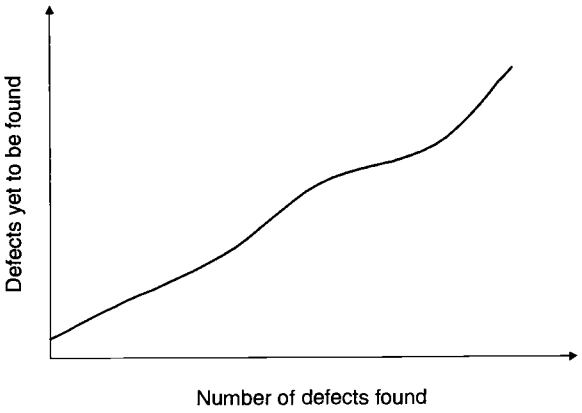
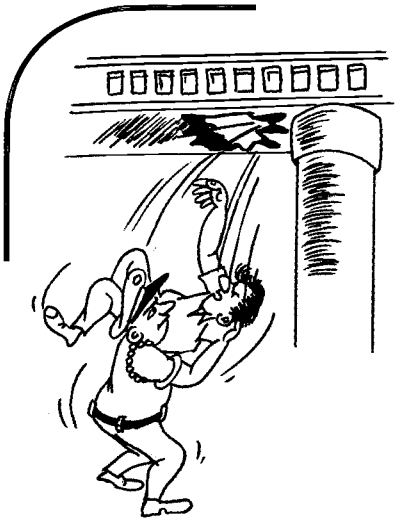
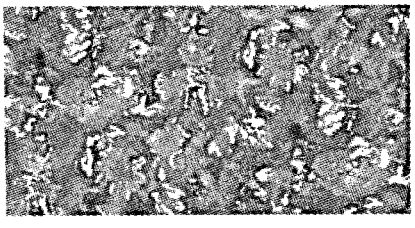


Figure 1.4
The number of defects yet to be found increases with the number of defects uncovered.

A fix for a defect is made around certain lines of code. This fix can produce side-effects around the same piece of code. This sets in spiraling changes to the program, all localized to certain select portions of the code. When we look at the code that got the fixes for the convoy of defects, it is likely to look like a piece of rag! Fixing a tear in one place in a shirt would most likely cause damage in another place. The only long-term solution in such a case is to throw away the shirt and create a new one. This amounts to a re-architecting the design and rewriting the code.

1.10 THE POLICEMEN ON THE BRIDGE





There was a wooden bridge on top of a river in a city. Whenever people walked over it to cross the river, they would fall down. To take care of this problem, the city appointed a strong policeman to stand under the bridge to save people who fall down. While this helped the problem to some extent, people continued to fall down the bridge. When the policeman moved to a different position, a new policeman was appointed to the job. During the first few days, instead of standing at the bottom of the bridge and saving the falling people, the new policeman worked with an engineer and fixed the hole on the bridge, which had not been noticed by the earlier policeman. People then stopped falling down the bridge and the new policeman did not have anyone to save. (This made his current job redundant and he moved on to do other things that yielded even better results for himself and the people...)

Prevention is better than cure—you may be able to expand your horizon much farther.

Testers are probably best equipped to know the problems customers may encounter. Like the second police officer in the above story, they know people fall and they know why people fall. Rather than simply catch people who fall (and thereby be exposed to the risk of a missed catch), they should also look at the root cause for falling and advise preventive action. It may not be possible for testers themselves to carry out preventive action. Just as the second police officer had to enlist the help of an engineer to plug the hole, testers would have to work with development engineers to make sure the root cause of the defects are addressed. The testers should not feel that by eliminating the problems totally their jobs are at stake. Like the second policeman, their careers can be enriching and beneficial to the organization if they harness their defect detection experience and transform some of it to defect prevention initiatives.

Defect prevention is a part of a tester's job. A career as a tester can be enriching and rewarding, if we can balance defect prevention and defect detection activities. Some of these career path possibilities are encapsulated in a three-stage model in Chapter 13, Common People Issues. We will now visit the question of what is the right balance between defect prevention and defect detection.

1.11 THE ENDS OF THE PENDULUM

The eventual goal of any software organization is to ensure that the customers get products that are reasonably free of defects. There are two approaches to achieving this goal. One is to focus on defect detection and correction and the second is to focus on defect prevention. These are also called *quality control* focus and *quality assurance* focus.

Testing is traditionally considered as a quality control activity, with an emphasis on defect detection and correction. We would like to take a broader view of testing and believe that there are aspects of testing that are also defect prevention oriented. For example, one of the aspects of white box testing, discussed in Chapter 3, is static testing, that involves desk checking, code walkthroughs, code reviews, and inspection. Even though these are traditionally considered as "quality assurance" activities, planning for overall testing activities with an intent to deliver quality products to customers, cannot be done effectively unless we take a holistic view of what can be done using quality assurance and what can be done with quality control (or the traditional definition of testing).

Quality assurance is normally associated with process models such as CMM, CMMI, ISO 9001, and so on. Quality control, on the other hand, is associated with testing (that form the bulk of the discussions in this book). This has caused an unnatural dichotomy between these two functions. Unfortunately, organizations view these two functions as mutually exclusive, "either–or" choices. We have even heard statements such as "with good processes, testing becomes redundant" or "processes are mere overheads—we

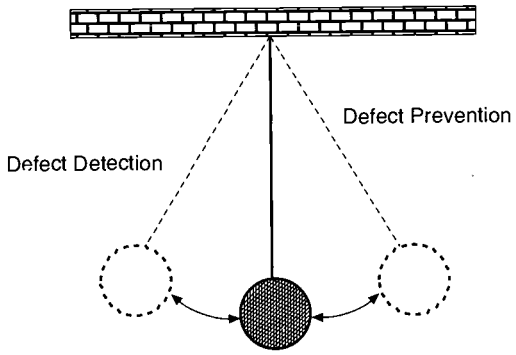


Figure 1.5
Quality control and quality assurance as two methods to achieve quality.

can find out everything by testing." It is almost as if there are two schools of thought at either extremes of a pendulum—one rooting for defect prevention (quality assurance) focus and the other rooting for the defect detection (quality control) focus. It is also common to find an organization swinging from one extreme to another over time, like a pendulum (Figure 1.5).

Rather than view defect prevention and defect detection as mutually exclusive functions or ends of a pendulum, we believe it is worthwhile to view these two as supplementary activities, being done in the right mix. Figure 1.6 gives a defect prevention—defect detection grid, which views the two functions as two dimensions. The right mix of the two activities corresponds to choosing the right quadrant in this grid.

When the focus on defect prevention is low, the emphasis on the use of appropriate standards, reviews, and processes are very low. This acts as an ideal "breeding ground" for defects. Most of the effort in ensuring quality of a product is left in the hands of the testing and defect detection team. If the focus on defect detection is also low (represented by the lower left-hand quadrant), this is a bad state for an organization to be in. Lack of testing and

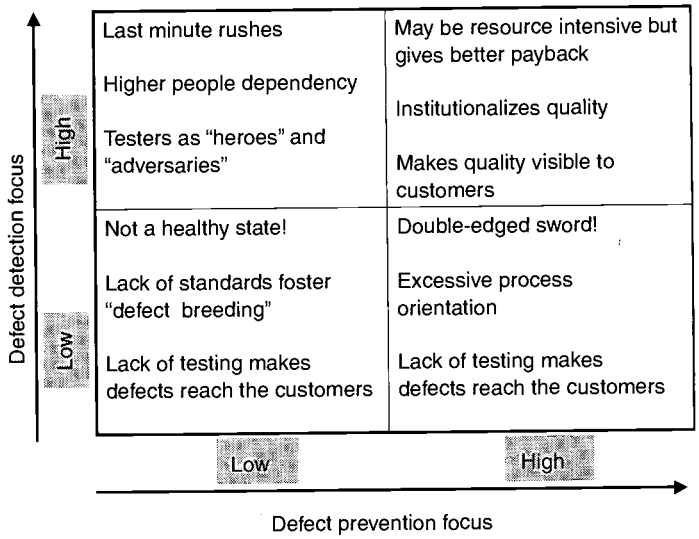
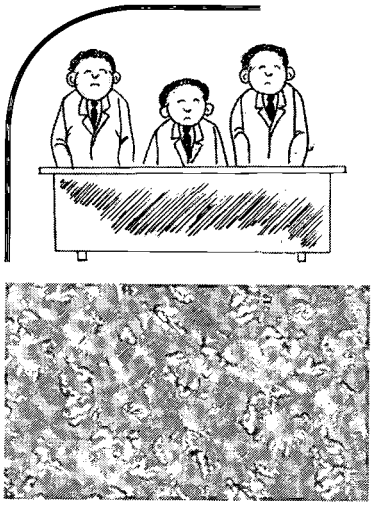


Figure 1.6
Relationship between defect detection focus and defect prevention focus.

defect detection activities does not "kill" these defects in time; hence the defects reach the customers. This is obviously not a healthy state to be in.

Even when the defect detection focus increases, with continued low defect prevention focus (upper left hand quadrant), the testing functions become a high-adrenalin rush, high-pressure job. Most defects are detected in the last minute—before the product release. Testers thus become superheroes who "save the day" by finding all the defects just in time. They may also become adversaries to developers as they always seem to find problems in what the developers do. This quadrant is better than the previous one, but ends up being difficult to sustain because the last-minute adrenalin rush burns people out faster.



Three Chinese doctors were brothers. The youngest one was a surgeon and well known in all parts of the world. He could find tumors in the body and remove them. The middle one was a doctor who could find out disease in its early days and prescribe medicine to cure it. He was known only in the city they lived in. The eldest of the brothers was not known outside the house, but his brothers always took his advice because he was able to tell them how to prevent any illness before they cropped up. The eldest brother may not have been the most famous, but he was surely the most effective.

Preventing an illness is more effective than curing it. People who prevent defects usually do not get much attention. They are usually the unsung heroes of an organization. Those who put out the fires are the ones who get visibility, not necessarily those who make sure fires do not happen in the first place. This, however, should not deter the motivation of people from defect prevention.

As we saw in the previous section, defect prevention and defect detection are not mutually exclusive. They need to be balanced properly for producing a quality product. Defect prevention improves the quality of the process producing the products while defect detection and testing is needed to catch and correct defects that escape the process. Defect prevention is thus process focused while defect detection is product focused. Defect detection acts as an extra check to augment the effectiveness of defect prevention.

An increase in defect prevention focus enables putting in place review mechanisms, upfront standards to be followed, and documented processes for performing the job. This upfront and proactive focus on doing things right to start with causes the testing (or defect detection) function to add more value, and enables catching any residual defects (that escape the defect prevention activities) before the defects reach the customers. Quality is institutionalized with this consistently high focus on both defect prevention

Defect prevention and defect detection should supplement each other and not be considered as mutually exclusive.

and defect detection. An organization may have to allocate sufficient resources for sustaining a high level of both defect prevention and defect detection activities (upper right-hand quadrant in Figure 1.6).

However, an organization should be careful about not relying too much on defect prevention and reducing the focus on defect detection (lower right-hand quadrant in Figure 1.6). Such a high focus on defect prevention and low focus on defect detection would not create a feeling of comfort amongst the management on the quality of product released since there are likely to be minimal internal defects found. This feeling will give rise to introduction of new processes to improve the effectiveness of defect detection. Too much of processes and such defect prevention initiatives may end up being perceived as a bureaucratic exercise, not flexible or adaptable to different scenarios. While processes bring in discipline and reduce dependency on specific individuals, they—when not implemented in spirit—could also end up being double-edged swords, acting as a damper to people's drive and initiative. When an organization pays equally high emphasis to defect prevention and defect detection (upper right corner in the grid), it may appear that it is expensive but this investment is bound to have a rich payback by institutional quality internally and making the benefits visible externally to the customers.

An organization should choose the right place on each of these two—defect detection and defect prevention—dimensions and thus choose the right place in the grid. The relative emphasis to be placed on the two dimensions will vary with the type of product, closeness to the release date, and the resources available. Making a conscious choice of the balance by considering the various factors will enable an organization to produce better quality products. It is important for an organization not to over-emphasize one of these at the expense of the other, as the next section will show.

1.12 MEN IN BLACK

Pride in "test" will take care of the "rest!"

As we can see from all the above discussions, testing requires abundant talent in multiple dimensions. People in the testing profession should have a customer focus, understanding the implications from the customer's perspective. They should have adequate analytical skills to be able to choose the right subset of tests and be able to counter the pesticide paradox. They should think ahead in terms of defect prevention and yet be able to spot and rectify errors that crop up. Finally (as we will see in the next section), they must be able to perform automation functions.

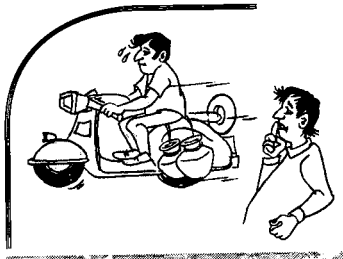
Despite all these challenging technical and inter-personal skills required, testing still remains a not-much-sought-after function. There was an interesting experiment that was described by De Marco and Lister in their book, *Peopleware* [DEMA–1987]. The testing team was seeded with motivated people who were "free from cognitive dissonance that hampers developers when testing their own programs." The team was given an identity (by a black dress, amidst the traditionally dressed remainder of the organization)

and tremendous importance. All this increased their pride in work and made their performance grow by leaps and bounds, "almost like magic." Long after the individual founding members left and were replaced by new people, the "Black Team" continued its existence and reputation.

The biggest bottleneck in taking up testing as a profession is the lack of self-belief. This lack of self-belief and apparent distrust of the existence of career options in testing makes people view the profession as a launching pad to do other software functions (notably, "development," a euphemism for coding). As a result, testers do not necessarily seek a career path in testing and develop skepticism towards the profession.

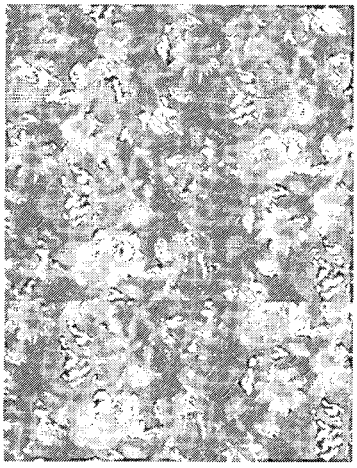
We have devoted an entire chapter in Part III of the book to career aspirations and other similar issues that people face. A part of the challenge that is faced is the context of globalization—the need to harness global resources to stay competitive. We address the organizational issues arising out of this in another chapter in Part III.

1.13 AUTOMATION SYNDROME



A farmer had to use water from a well which was located more than a mile away. Therefore, he employed 100 people to draw water from the well and water his fields. Each of those employed brought a pot of water a day but this was not sufficient. The crops failed.

Just before the next crop cycle, the farmer remembered the failures of the previous season. He thought about automation as a viable way to increase productivity and avoid such failures. He had heard about motorcycles as faster means of commuting (with the weight of water). Therefore, he got 50 motorcycles, laid off 50 of his workers and asked each rider to get two pots of water.Apparently, the correct reasoning was that thanks to improved productivity (that is, speed and convenience of a motorcycle), he needed fewer people. Unfortunately, he choose to use motorcycles just before his crop cycle started. Hence for the first few weeks, the workers were kept busy learning to use the motorcycle. In the process of learning to balance the motorcycles, the number of pots of water they could fetch fell. Added to this, since the num-



ber of workers was also lower, the productivity actually dropped. The crops failed again.

The next crop cycle came. Now all workers were laid off except one. The farmer bought a truck this time to fetch water. This time he realized the need for training and got his worker to learn driving. However, the road leading to the farm from the well was narrow and the truck did not help in bringing in the water. No portion of the crop could be saved this time also.

After these experiences the farmer said, "My life was better without automation!"

Failures outnumber successes in automation. Equal skills and focus are needed for automation as in product development.

If you go through the story closely there appear to be several reasons for the crop failures that are not to do with the automation intent at all. The frustration of the farmer should not be directed at automation but on the process followed for automation and the inappropriate choices made. In the second crop cycle, the reason for failure was lack of skills and in the third cycle it is due to improper tool implementation.

In the first crop cycle, the farmer laid off his workers immediately after the purchase of motorcycles and expected cost and time to come down. He repeated the same mistake for the third crop cycle. Automation does not yield results immediately.

The moral of the above story as it applies to testing is that automation requires careful planning, evaluation, and training. Automation may not produce immediate returns. An organization that expects immediate returns from automation may end up being disappointed and wrongly blame automation for their failures, instead of objectively looking at their level of preparedness for automation in terms of planning, evaluation, and training.

A large number of organizations fail in their automation initiatives and revert to manual testing. Unfortunately, they conclude—wrongly—that automation will never work.

Testing, by nature, involves repetitive work. Thus, it lends itself naturally to automation. However, automation is a double-edged sword. Some of the points that should be kept in mind while harping on automation are as follows.

- Know first why you want to automate and what you want to automate, before recommending automation for automation's sake.
- Evaluate multiple tools before choosing one as being most appropriate for your need.
- Try to choose tools to match your needs, rather than changing your needs to match the tool's capabilities.

- Train people first before expecting them to be productive.
- Do not expect overnight returns from automation.

1.14 PUTTING IT ALL TOGETHER

We have discussed several basic principles of testing in this chapter. These principles provide an anchor to the other chapters that we have in rest of the book. We have organized the book into five parts. The first part (which includes this chapter) is *Setting the Context*, which sets the context for the rest of the book. In the chapter that follows, we cover Software Development Life Cycle (SDLC) Models in the context of testing, verification and validation activities.

In Part II, *Types of Testing*, we cover the common types of testing. Chapters 3 through 10 cover white box testing, black box testing, integration testing, system and acceptance testing, performance testing, regression testing, internationalization testing, and ad hoc testing.

Part III, *Select Topics in Specialized Testing*, addresses two specific and somewhat esoteric testing topics—object oriented testing in Chapter 11 and usability and accessibility testing in Chapter 12.

Part IV, *People and Organizational Issues in Testing*, provides an oftignored perspective. Chapter 13 addresses the common people issues like misconceptions, career path concerns and so on. Chapter 14 address the different organizational structures in vogue to set up effective testing teams, especially in the context of globalization.

The final part, Part V, *Test Management and Automation*, addresses the process, management, and automation issues to ensure effective testing in an organization. Chapter 16 discusses test planning management and execution. This discusses various aspects of putting together a test plan, tracking a testing project and related issues. Chapter 17 goes into details of the benefits, challenges, and approaches in test automation—an area of emerging and increasing importance in the test community. The final chapter, Chapter 18, goes into details of what data are required to be captured and what analysis is to be performed for measuring effectiveness of testing, quality of a product and similar perspectives and how this information can be used to achieve quantifiable continuous improvement.

While we have provided the necessary theoretical foundation in different parts of the book, our emphasis throughout the book has been on the state of practice. This section should set the context for what the reader can expect in rest of the book.

REFERENCES

One of the early seminal works on testing is [MYER-79]. In particular, the example of trying to write test cases for verifying three numbers to be the sides of a valid triangle still remains one of the best ways to bring forth the principles of testing. [DEMA-87] provides several interesting perspectives of the entire software engineering discipline. The concept of black team has been illustrated in that work. The emphasis required for process and quality assurance methodologies and the balance to be struck between quality assurance and quality control are brought out in [HUMP-86]. Some of the universally applicable quality principles are discussed in the classics [CROS-80] and [DEMI-86]. [DIJK-72], a Turing Award lecture brings out the doctrine of program testing can never prove the absence of defects. [BEIZ-90] discusses the pesticide paradox.

PROBLEMS AND EXERCISES

1. We have talked about the pervasiveness of software as a reason why defects left in a product would get detected sooner than later. Assume that televisions with embedded software were able to download, install and self-correct patches over the cable network automatically and the TV manufacturer told you that this would just take five minutes every week "at no cost to you, the consumer." Would you agree? Give some reasons why this is not acceptable.
2. Your organization has been successful in developing a client-server application that is installed at several customer locations. You are changing the application to be a hosted, web-based application that anyone can use after a simple registration process. Outline some of the challenges that you should expect from a quality and testing perspective of the changed application.
3. The following were some of the statements made by people in a product development organization. Identify the fallacies if any in the statements and relate it to the principles discussed in this chapter.
    a. "The code for this product is generated automatically by a CASE tool – it is therefore defect – free."
    b. "We are certified according to the latest process models – we do not need testing."
    c. "We need to test the software with dot matrix printers because we have never released a product without testing with a dot matrix printer."

d. "I have run all the tests that I have been running for the last two releases and I don't need to run any more tests."

e. "This automation tool is being used by our competitors – hence we should also use the same tool."

4. Assume that each defect in gathering requirements allowed to go to customers costs $10,000, and that the corresponding costs for design defects and coding defects are $1,000 and $100, respectively. Also, assume that current statistics indicate that on average ten new defects come from each of the phases. In addition, each phase also lets the defects from the previous phase seep through. What is the total cost of the defects under the current scenario? If you put a quality assurance process to catch 50% of the defects from each phase not to go to the next phase, what are the expected cost savings?

5. You are to write a program that adds two two-digit integers. Can you test this program exhaustively? If so, how many test cases are required? Assuming that each test case can be executed and analyzed in one second, how long would it take for you to run all the tests?

6. We argued that the number of defects left in a program is proportional to the number of defects detected. Give reasons why this argument looks counterintuitive. Also, give practical reasons why this phenomenon causes problems in testing.

Software Development Life Cycle Models

CHAPTER 2

In this chapter—

- ✓ Phases of software project
- ✓ Quality, quality assurance, and quality control
- ✓ Testing, verification, and validation
- ✓ Process model to represent different phases
- ✓ Life cycle models

2.1 PHASES OF SOFTWARE PROJECT

A software project is made up of a series of phases. Broadly, most software projects comprise the following phases.

- Requirements gathering and analysis
- Planning
- Design
- Development or coding
- Testing
- Deployment and maintenance

2.1.1 Requirements Gathering and Analysis

During requirements gathering, the specific requirements of the software to be built are gathered and documented. If the software is bespoke software, then there is a single customer who can give these requirements. If the product is a general-purpose software, then a product marketing team within the software product organization specifies the requirements by aggregating the requirements of multiple potential customers. In either case, it is important to ensure that the right requirements are captured at every stage. The requirements get documented in the form of a System Requirements Specification (SRS) document. This document acts as a bridge between the customer and the designers chartered to build the product.

2.1.2 Planning

The purpose of the planning phase is to come up with a schedule, the scope, and resource requirements for a release. A plan explains how the requirements will be met and by which time. It needs to take into account the requirements—what will be met and what will not be met—for the current release to decide on the scope for the project, look at resource availability, and to come out with set of milestones and release date for the project. The planning phase is applicable for both development and testing activities. At the end of this phase, both project plan and test plan documents are delivered.

2.1.3 Design

The purpose of the design phase is to figure out how to satisfy the requirements enumerated in the System Requirements Specification document. The design phase produces a representation that will be used by the following phase, the development phase. This representation should serve two purposes. First, from this representation, it should be possible to verify that all the requirements are satisfied. Second, this representation

should give sufficient information for the development phase to proceed with the coding and implementation of the system. Design is usually split into two levels—high-level design and low-level or a detailed design. The design step produces the system design description (SDD) document that will be used by development teams to produce the programs that realize the design.

2.1.4 Development or Coding

Design acts as a blueprint for the actual coding to proceed. This development or coding phase comprises coding the programs in the chosen programming language. It produces the software that meets the requirements the design was meant to satisfy. In addition to programming, this phase also involves the creation of product documentation.

2.1.5 Testing

As the programs are coded (in the chosen programming language), they are also tested. In addition, after the coding is (deemed) complete, the product is subjected to testing. Testing is the process of exercising the software product in pre-defined ways to check if the behavior is the same as expected behavior. By testing the product, an organization identifies and removes as many defects as possible before shipping it out.

2.1.6 Deployment and Maintenance

Once a product is tested, it is given to the customers who deploy it in their environments. As the users start using the product in their environments, they may observe discrepancies between the actual behavior of the product and what they were given to expect (either by the marketing people or through the product documentation). Such discrepancies could end up as product defects, which need to be corrected. The product now enters the maintenance phase, wherein the product is maintained or changed to satisfy the changes that arise from customer expectations, environmental changes, etc. Maintenance is made up of *corrective maintenance* (for example, fixing customer-reported problems), *adaptive maintenance* (for example, making the software run on a new version of an operating system or database), and *preventive maintenance* (for example, changing the application program code to avoid a potential security hole in an operating system code).

2.2 QUALITY, QUALITY ASSURANCE, AND QUALITY CONTROL

A software product is designed to satisfy certain requirements of a given customer (or set of customers). How can we characterize this phrase—"satisfying requirements"? Requirements get translated into software features, each feature being designed to meet one or more of the

Quality is meeting the requirements expected of the software, consistently and predictably.

requirements. For each such feature, the *expected behavior* is characterized by a set of *test cases*. Each test case is further characterized by

1. The environment under which the test case is to be executed;
2. Inputs that should be provided for that test case;
3. How these inputs should get processed;
4. What changes should be produced in the internal state or environment; and
5. What outputs should be produced.

The *actual behavior* of a given software for a given test case, under a given set of inputs, in a given environment, and in a given internal state is characterized by

1. How these inputs actually get processed;
2. What changes are actually produced in the internal state or environment; and
3. What outputs are actually produced.

If the actual behavior and the expected behavior are identical in all their characteristics, then that test case is said to be passed. If not, the given software is said to have a *defect* on that test case.

How do we increase the chances of a product meeting the requirements expected of it, consistently and predictably? There are two types of methods—quality control and quality assurance.

Quality control attempts to build a product, test it for expected behavior after it is built, and if the expected behavior is not the same as the actual behavior of the product, fixes the product as is necessary and rebuilds the product. This iteration is repeated till the expected behavior of the product matches the actual behavior for the scenarios tested. Thus quality control is defect-detection and defect-correction oriented, and works on the product rather than on the process.

Quality assurance, on the other hand, attempts defect prevention by concentrating on the process of producing the product rather than working on defect detection/correction after the product is built. For example, instead of producing and then testing a program code for proper behavior by exercising the built product, a quality assurance approach would be to first review the design before the product is built and correct the design errors in the first place. Similarly, to ensure the production of a better code, a quality assurance process may mandate coding standards to be followed by all programmers. As can be seen from the above examples, quality assurance normally tends to apply to all the products that use a process. Also, since quality assurance continues throughout the life of the product it is everybody's responsibility; hence it is a staff function. In contrast, the responsibility for quality control is usually localized to a quality control team. Table 2.1 summarizes the key distinctions between quality control and quality assurance.

Table 2.1. Difference between quality assurance and quality control.

Quality Assurance	Quality Control
Concentrates on the process of producing the products	Concentrates on specific products
Defect-prevention oriented	Defect-detection and correction oriented
Usually done throughout the life cycle	Usually done after the product is built
This is usually a staff function	This is usually a line function
Examples: reviews and audits	Examples: software testing at various levels

We will see more details of quality assurance methods such as reviews and audits in Chapter 3. But the focus of the rest of this book is on software testing, which is essentially a quality control activity. Let us discuss more about testing in the next section.

2.3 TESTING, VERIFICATION, AND VALIDATION

Verification is the process of evaluating a system or component to determine whether the products of a given phase satisfy the conditions imposed at the start of that phase.

The narrow definition of the term "testing" is the phase that follows coding and precedes deployment. Testing is traditionally used to mean testing of the program code. However, coding is a downstream activity, as against requirements and design that occur much earlier in a project life cycle. Given that the objective of a software project is to minimize and prevent defects, testing of program code alone is not sufficient. As we saw in the last chapter, defects can creep in during any phase and these defects should be detected as close to the point of injection as possible and not wait till the testing of programs. Hence against this, if each phase is "tested" separately as and when the phase is completed (or, better still, as the phase is being executed), then defects can be detected early, thereby reducing the overall costs.

Timely testing increases the chances of a product or service meeting the customer's requirements. When a product is tested with appropriate and realistic tests that reflect typical usage patterns by the intended users, the chances of the product satisfying the customer's requirement is much higher. While testing does not guarantee zero defects, effective testing certainly increases the chances of customer acceptance of the software.

The purpose of testing is to uncover defects in the system (and to have someone fix the defects). Testing is done by a set of people within a software product (or service) organization whose goal and charter is to uncover the defects in the product before it reaches the customer (see Section 1.3). As we saw in the previous chapter, the purpose of testing is NOT to prove that the product has no defects. The purpose of software testing is to find defects in a software product. As we will see in the chapters on people and organizational

Validation is the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.

issues (Chapters 13, 14), the reward systems and the organization structures should create and foster an environment that encourages this purpose of testing.

Testing is NOT meant to replace other ways of ensuring quality (like reviews). It is one of the methods to detect defects in a software product. There are other methods that achieve the same function. For example, we will see later that following well-defined processes and standards reduces the chances of defects creeping into a software. We will also discuss other methods like reviews and inspections, which actually attempt to prevent defects coming into the product. To be effective, testing should complement, supplement, and augment such quality assurance methods discussed in the previous section.

The idea of catching defects within each phase, without letting them reach the testing phase, leads us to define two more terms—verification and validation.

During the requirements gathering phase, the requirements are faithfully captured. The SRS document is the product of the requirements phase. To ensure that requirements are faithfully captured, the customer verifies this document. The design phase takes the SRS document as input and maps the requirements to a design that can drive the coding. The SDD document is the product of the design phase. The SDD is verified by the requirements team to ensure that the design faithfully reflects the SRS, which imposed the conditions at the beginning of the design phase.

Verification takes care of activities to focus on the question *"Are we building the product right?"* and validation takes care of a set of activities to address the question *"Are we building the right product?"*

To build the product right, certain activities/conditions/procedures are imposed at the beginning of the life cycle. These activities are considered *"proactive"* as their purpose is to prevent the defects before they take shape. The process activities carried out during various phases for each of the product releases can be termed as verification. Requirements review, design review, and code review are some examples of verification activities.

To build the right product, certain activities are carried out during various phases to validate whether the product is built as per specifications. These activities are considered *"reactive"* as their purpose is to find defects that affect the product and fix them as soon as they are introduced. Some examples of validation include unit testing performed to verify if the code logic works, integration testing performed to verify the design, and system testing performed to verify that the requirements are met.

Quality Assurance = Verification
Quality Control = Validation = Testing

To summarize, there are different terminologies that may stand for the same or similar concepts. For all practical purposes in this book, we can assume verification and quality assurance to be one and the same. Similarly quality control, validation, and testing mean the same.

2.4 PROCESS MODEL TO REPRESENT DIFFERENT PHASES

A process model is a way to represent any given phase of software development that effectively builds in the concepts of validation and verification to prevent and minimize the delay between defect injection and defect detection (and eventual correction). In this model, each phase of a software project is characterized by the following.

- Entry criteria, which specify when that phase can be started. Also included are the inputs for the phase.
- Tasks, or steps that need to be carried out in that phase, along with measurements that characterize the tasks.
- Verification, which specifies methods of checking that the tasks have been carried out correctly.
- Exit criteria, which stipulate the conditions under which one can consider the phase as done. Also included are the outputs for only the phase.

This model, known as the **E**ntry **T**ask **V**erification e**X**it or ETVX model, offers several advantages for effective verification and validation.

1. Clear entry criteria make sure that a given phase does not start prematurely.
2. The verification for each phase (or each activity in each phase) helps prevent defects, or at least, minimizes the time delay between defect injection and defect detection.
3. Documentation of the detailed tasks that comprise each phase reduces the ambiguity in interpretation of the instructions and thus minimizes the variations that can come from repeated executions of these tasks by different individuals.
4. Clear exit criteria provide a means of validation of the phase, after the phase is done but before handing over to the next phase.

An example of applying the ETVX model to the design phase is presented in Figure 2.1.

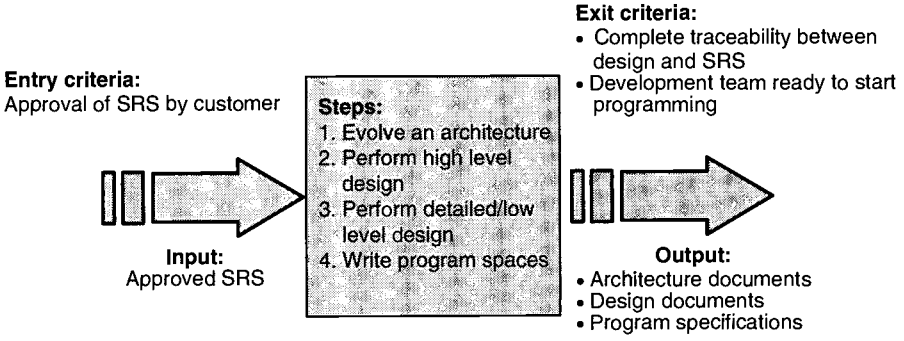


Figure 2.1
ETVX model applied to design.

2.5 LIFE CYCLE MODELS

The ETVX model characterizes a phase of a project. A Life Cycle model describes how the phases combine together to form a complete project or life cycle. Such a model is characterized by the following attributes.

The activities performed In any given software project, apart from the most common activities or phases—requirements gathering, design, development, testing, and maintenance—there could be other activities as well. Some of these activities could be technical activities (for example, porting) and some could be non-technical (for example, hiring).

The deliverables from each activity Each activity produces a set of deliverables, which are the end products of that activity. For example, the requirements gathering phase produces the SRS document, the design phase produces the SDD document, and so on.

Methods of validation of the deliverables The outputs produced by a given activity represent the goal to be satisfied by that activity. Hence it is necessary to have proper validation criteria for each output.

The sequence of activities The different activities work together in unison in a certain sequence of steps to achieve overall project goals. For example, the process of requirements gathering may involve steps such as interviews with customers, documentation of requirements, validation of documented requirements with customers, and freezing of requirements. These steps may be repeated as many times as needed to get the final frozen requirements.

Methods of verification of each activity, including the mechanism of communication amongst the activities The different activities interact with one another by means of communication methods. For example, when a defect is found in one activity and is traced back to the causes in an earlier activity, proper verification methods are needed to retrace steps from the point of defect to the cause of the defect.

We will now look at some of the common life cycle models that are used in software projects. For each model, we will look at:

1. a brief description of the model;
2. the relationship of the model to verification and validation activities; and
3. typical scenarios where that life cycle model is useful.

2.5.1 Waterfall Model

In the Waterfall model, a project is divided into a set of phases (or activities). Each phase is distinct, that is, there are clear lines of separation between the phases, with very clear demarcation of the functions of each of the phases.

A project starts with an initial phase, and upon completion of the phase, moves on to the next phase. On the completion of this phase, the project moves to the subsequent phase and so on. Thus the phases are strictly time sequenced.

We depict one example of a project in the Waterfall model in Figure 2.2. The project goes through a phase of requirements gathering. At the end of requirements gathering, a System Requirements Specification document is produced. This becomes the input to the design phase. During the design phase, a detailed design is produced in the form of a System Design Description. With the SDD as input, the project proceeds to the development or coding phase, wherein programmers develop the programs required to satisfy the design. Once the programmers complete their coding tasks, they hand the product to the testing team, who test the product before it is released.

If there is no problem in a given phase, then this method can work, going in one direction (like a waterfall). But what would happen if there are problems after going to a particular phase? For example, you go into the design phase and find that it is not possible to satisfy the requirements,

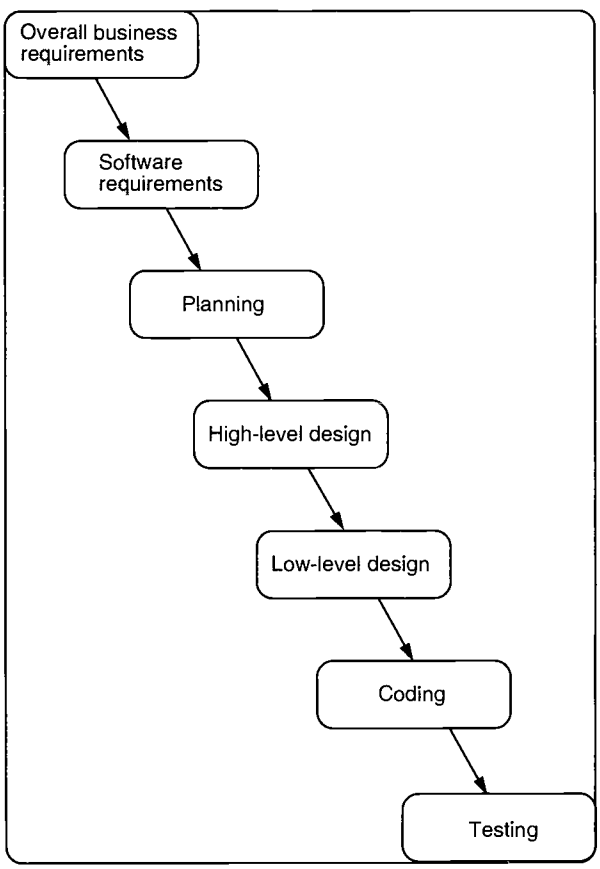


Figure 2.2
Waterfall model.

A Waterfall model is characterized by three attributes.

1. The project is divided into separate, distinct phases.
2. Each phase communicates to the next through pre-specified outputs.
3. When an error is detected, it is traced back to one previous phase at a time, until it gets resolved at some earlier phase.

going by the current design approach being used. What could be the possible causes and remedies? You may try an alternative design if possible and see if that can satisfy the requirements. If there are no alternative design approaches possible, then there must be feedback to the requirements phase to correct the requirements.

Let us take the example one step further. Suppose a design was created for a given set of requirements and the project passed on to the programming/development phase. At this point of time, it was found that it was not possible to develop the programs because of some limitations. What would you do? One approach would be to try out alternative strategies in the development phase so that the design could still be satisfied. Another possibility could be that there are flaws in design that cause conflicts during development and hence the design has to be revisited. When the design phase is revisited—like in the previous case—it may happen that the problem may have to be addressed in the requirements phase itself. So, a problem in one phase could potentially be traced back to *any* of the previous phases.

Since each phase has an output, the latter can be validated against a set of criteria. To increase the effectiveness, the completion criteria for each output can be published a priori. Before a phase starts, the completion criteria for the previous phase can be checked and this can act as a verification mechanism for the phase. This can minimize the kind of delays we discussed in the example above.

The main strength of the Waterfall Model is its simplicity. The model is very useful when a project can actually be divided into watertight compartments. But very few software projects can be divided thus. The major drawback in the Waterfall model arises from the delay in feedback among the phases, and thus the ineffectiveness of verification and validation activities. An error in one phase is not detected till at least the next phase. When a given phase detects an error, the communication is only to the immediately preceding phase. This sequential nature of communication among the phases can introduce inordinate delays in resolving the problem. The reduced responsiveness that is inherent in the model and the fact that the segregation of phases is unrealistic severely restricts the applicability of this model.

2.5.2 Prototyping and Rapid Application Development Models

Prototyping and Rapid Application Development (RAD) models recognize and address the following issues.

1. Early and frequent user feedback will increase the chances of a software project meeting the customers' requirements.
2. Changes are unavoidable and the software development process must be able to adapt itself to rapid changes.

1. A Prototyping model uses constant user interaction, early in the requirements gathering stage, to produce a prototype.
2. The proto-type is used to derive the system requirements specification and can be discarded after the SRS is built.
3. An appropriate life cycle model is chosen for building the actual product after the user accepts the SRS.

The Prototyping model comprises the following activities.

1. The software development organization interacts with customers to understand their requirements.
2. The software development organization produces a prototype to show how the eventual software system would look like. This prototype would have the models of how the input screens and output reports would look like, in addition to having some "empty can functionality" to demonstrate the workflow and processing logic.
3. The customer and the development organization review the prototype frequently so that the customer's feedback is taken very early in the cycle (that is, during the requirements gathering phase).
4. Based on the feedback and the prototype that is produced, the software development organization produces the System Requirements Specification document.
5. Once the SRS document is produced, the prototype can be discarded.
6. The SRS document is used as the basis for further design and development.

Thus, the prototype is simply used as a means of quickly gathering (the right) requirements. This model has built-in mechanisms for verification and validation of the requirements. As the prototype is being developed, the customer's frequent feedback acts as a validation mechanism. Once the SRS is produced, it acts as the verification mechanism for the design and subsequent steps. But the verification and validation activities of the subsequent phases are actually dictated by the life cycle model that is followed after the SRS is obtained.

This model is obviously advantageous when a customer can participate by giving feedback. This model is also useful in cases where the feedback can be easily quantified and incorporated, for example, determining user interface, predicting performance, and so on.

For a general-purpose product, which is meant for many customers, there is no single customer whose feedback can be taken as final. In these cases, a product manager in the marketing group of the product vendor usually plays the role of the eventual customer. Hence the applicability of this model is somewhat limited to general-purpose products. Furthermore, the prototype is used as a means of capturing requirements and is not necessarily meant to be used afterwards. Oftentimes, the prototype (or parts of the prototype) makes its way to becoming the product itself. This can have undesirable effects as the prototype usually employs several short cuts, unstructured methods, and tools to achieve a quick turnaround. Such short cuts are potential sources of defects in live environments and thus can place a heavy burden on maintenance and testing.

The Rapid Application Development model is a variation of the Prototyping Model. Like the Prototyping Model, the RAD Model relies on feedback and interaction by the customers to gather the initial requirements.

However, the Prototyping model differs from the RAD Model on two counts.

First, in the RAD Model, it is not a prototype that is built but the actual product itself. That is, the built application (prototype, in the previous model) is not discarded. Hence, it is named Rapid Application Development model.

Second, in order to ensure formalism in capturing the requirements and proper reflection of the requirements in the design and subsequent phases, a Computer Aided Software Engineering (CASE) tool is used throughout the life cycle, right from requirements gathering. Such CASE tools have

- methodologies to elicit requirements;
- repositories to store the gathered requirements and all downstream entities such as design objects; and
- mechanisms to automatically translate the requirements stored in the repositories to design and generate the code in the chosen programming environment.

The methodologies provided by a CASE tool can provide inbuilt means of verification and validation. For example, the tool may be able to automatically detect and resolve inconsistencies in data types or dependencies. Since the design (and, perhaps, even the program code) can be automatically generated from the requirements, the validation can be very complete, extending to all the downstream phases, unlike the Prototyping model.

This method can have wider applicability for even general-purpose products. The automatic generation of the design and programs produced by a CASE tool makes this model more attractive. The cost of such CASE tools is a factor that an organization would have to consider before deciding on the use of this model for a given project. In addition, CASE tools and this model is generally more suited for applications projects rather than systems type projects.

2.5.3 Spiral or Iterative Model

The Spiral or Iterative model follows a process in which the requirements gathering, design, coding, and testing are performed iteratively till all requirements are met. There is also a good amount of overlap among the activities of requirements gathering, design, coding, and testing following this model. What phase the product is in is difficult to conclude as each requirement can be at a different phase. The only conclusion that can be made is at what phase *each* of the requirements is in. If a defect is produced in any phase of a given requirement, it may cause that requirement to revisit an earlier phase. This model enables incremental development whereby the product evolves, with requirements getting added to it dynamically. This enables the product to be demonstrated, at any point of time, with the functionality available at that point of time. It also enables the "increments" to be sent to the customer for approval. The progress of the product can be

Table 2.2 Some product requirements and phases.

Requirements	Status/Phase currently in
Requirement-1	Coding
Requirement-2	Design
Requirement-3	Requirement
Requirement-4	Testing
Requirement-5	Released

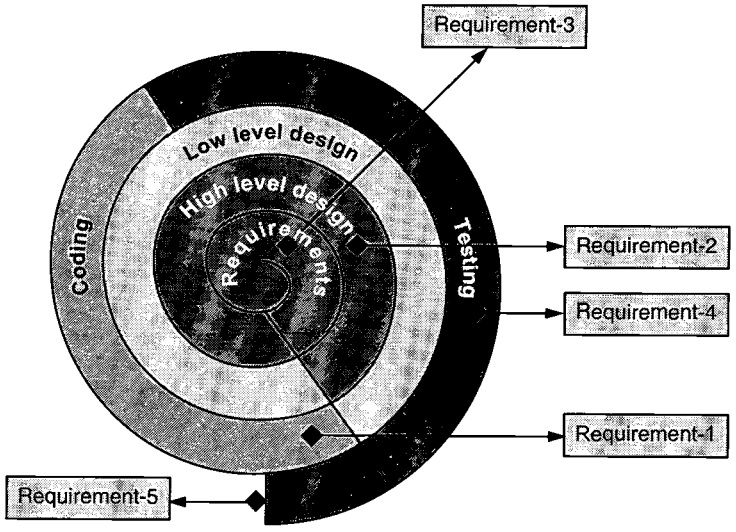


Figure 2.3
Spiral model.

seen from the beginning of the project as the model delivers "increments" at regular intervals. Even though it will be very difficult to plan a release date following this model, it allows the progress to be tracked and the customer approvals to be obtained at regular intervals, thereby reducing the risk of finding major defects at a later point of time. Table 2.2 gives an example of phases for some of the requirements in the product.

Figure 2.3 (the coloured figure is available on page 457) depicts the Spiral model and the phases involved in the model, for the example on Table 2.2. As can be seen, each requirement is "spiraling outwards" through the different phases as the entire project evolves.

2.5.4 The V Model

The Waterfall Model viewed testing as a post-development (that is, post-coding) activity. The Spiral Model took this one step further and tried to break up the product into increments each of which can be tested

separately. The V Model starts off being similar to the Waterfall Model in that it envisages product development to be made up of a number of phases or levels. However, the new perspective that the V Model brings in is that different types of testing apply at different levels. Thus, from a testing perspective, the type of tests that need to be done at each level vary significantly.

For instance, consider a typical product development activity represented as a Waterfall Model earlier in Figure 2.2. The system starts with the overall business requirements from the point of view of customers. These requirements cover hardware, software, and operational requirements. Since our focus is on the software, moving from overall requirements to software requirements becomes the next step. In order to realize the software requirements, the proposed software system is envisaged as a set of subsystems that work together. This high-level design (of breaking the system into subsystems with identified interfaces) then gets translated to a more detailed or low-level design. This detailed design goes into issues like data structures, algorithm choices, table layouts, processing logic, exception conditions, and so on. It results in the identification of a number of components, each component realized by program code written in appropriate programming languages.

Given these levels, what kind of tests apply in each of these levels? To begin with, for overall business requirements, eventually whatever software is developed should fit into and work in this overall context and should be accepted by the end users, in their environment. This testing, the final proof of the pudding, is *acceptance testing*. But, before the product is deployed in the customer's environment, the product vendor should test it as an entire unit to make sure that all the software requirements are satisfied by the product that is developed. This testing of the entire software system can be called *system testing*. Since high-level design views the system as being made up of interoperating and integrated (software) subsystems, the individual subsystems should be integrated and tested together before a full blown system test can be done. This testing of high-level design corresponds to *integration testing*. The components that are the outputs of the low-level design have to be tested independently before being integrated. Thus, the testing corresponding to the low-level design phase is *component testing*. Finally, since coding produces several program units, each of these smaller program units have to be tested independently before trying to combine them together to form components. This testing of the program units forms *unit testing*.

Figure 2.4 depicts the different types of testing that apply to each of the steps. For simplicity, we have not shown the planning phase as a separate entity since it is common for all testing phases. But, it is not possible to *execute* any of these tests until the product is actually built. In other words, the step called "testing" is now broken down into different sub-steps called acceptance testing, system testing, and so on as shown in Figure 2.4. So, it is still the case that all the testing *execution* related activities are done only at the end of the life cycle.

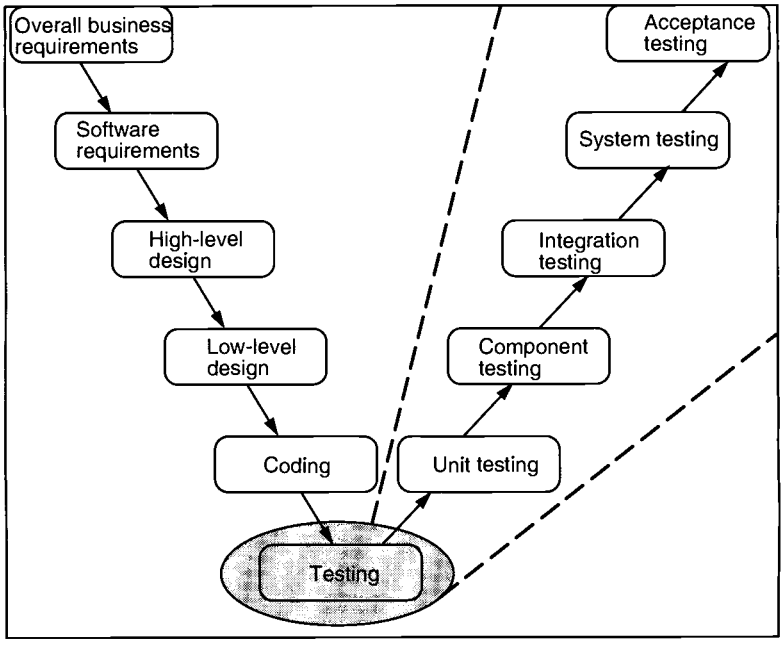


Figure 2.4
Phases of testing for different development phases.

1. The V-model splits testing into two parts—design and execution.
2. Test design is done early, while test execution is done in the end.
3. There are different types of tests for each phase of life cycle.

Even though the execution of the tests cannot be done till the product is built, the *design* of tests can be carried out much earlier. In fact, if we look at the aspect of skill sets required for *designing* each type of tests, the people best suited to design each of these tests are those who are actually performing the function of creating the corresponding artifact. For example, the best people to articulate what the acceptance tests should be are the ones who formulate the overall business requirements (and, of course, the customers, where possible). Similarly, the people best equipped to design the integration tests are those who know how the system is broken into subsystems and what the interfaces between the subsystems are—that is, those who perform the high-level design. Again, the people doing development know the innards of the program code and thus are best equipped to *design* the unit tests.

Not only are the skill sets required for designing these different types of tests different, but also, there is no reason to defer the designing of the tests till the very end. As and when each activity on the left-hand side of the "V" is being carried out, the design of the corresponding type of tests can be carried out. By performing an early design of the tests and deferring only the test execution till the end, we achieve three important gains.

- First, we achieve more parallelism and reduce the end-of-cycle time taken for testing.
- Second, by designing tests for each activity upfront, we are building in better upfront validation, thus again reducing last-minute surprises.
- Third, tests are designed by people with appropriate skill sets.

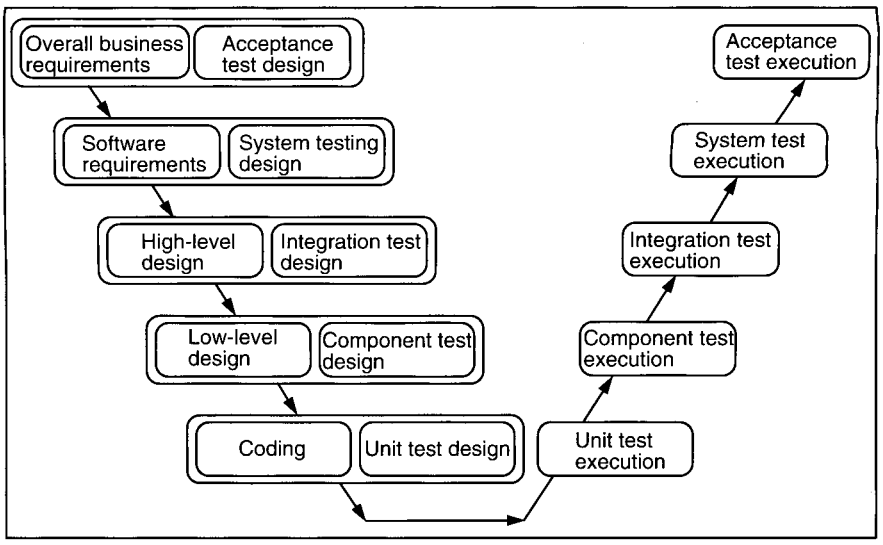


Figure 2.5
V Model.

This is the basis for the V Model, which presents excellent advantages for verification and validation. As shown in Figure 2.5, for each type of test, we move the design of tests upstream, along with the actual activities and retain the test execution downstream, after the product is built.

2.5.5 Modified V Model

The V Model split the design and execution portion of the various types of tests and attached the test design portion to the corresponding earlier phases of the software life cycle.

An assumption made there was that even though the activity of test execution was split into execution of tests of different types, the execution cannot happen until the entire product is built. For a given product, the different units and components can be in different stages of evolution. For example, one unit could be still under development and thus be in the unit-testing phase whereas another unit could be ready for component testing while the component itself may not be ready for integration testing. There may be components that are ready (that is, component tested) for integration and being subjected to integration tests (along with other modules which are also ready for integration, provided those modules can be integrated). The V Model does not explicitly address this natural parallelism commonly found in product development.

In the modified V Model, this parallelism is exploited. When each unit or component or module is given explicit exit criteria to pass on to the subsequent stage, the units or components or modules that satisfy a given phase of testing move to the next phase of testing where possible, without

1. The modified V model recognizes that different parts of a product are in different stages of evolution.
2. Each part enters the appropriate testing phase (such as unit testing, component testing, and so on) when the appropriate entry criteria are met.

necessarily waiting for all the units or components or modules to move in unison from one phase of testing to another, as shown in Figure 2.6.

Just as the V Model introduced various *types* of testing, the modified V model introduces various *phases* of testing. A phase of testing has a one-to-one mapping to the types of testing, that is, there is a unit-testing phase, component-testing phase, and so on. Once a unit has completed the unit-testing phase, it becomes part of a component and enters the component-testing phase. It then moves to integration-testing phase and so on. Rather than view the product as going through different types of tests (as the V model does), the modified V Model views each part of the product to go through different phases of testing. These are actually two sides of the same coin and thus provide complimentary views. The main advantage the modified V model brings to the table is the recognition of the parallelism present in different parts of the product and assigning each part to the most appropriate phase of testing that is possible. In Figure 2.6, the columns of the table represents one side of V, and rows (which are test phases) represent the other side of V.

In Figure 2.6, notice that different phases of testing are done in parallel. While starting a phase of testing it is important to look at whether the product is ready for testing. It is determined by a set of *entry criteria.* The earliest possible quality to start the next phase of testing is denoted by entry criteria, and to start the next phase of testing the earlier phase need not have completed. The testing phases are also associated with a set of exit criteria to complete the test activities for each phase. They are determined by *exit criteria.* The entry and exit criteria for each of the phases ensure that right quality of product delivered for starting the test and right amount of testing is completed for the release. Even though it is indicated in the picture all of the test phases finish at the same time, practically it can have different time lines. The longest phase determines the release date.

In Figure 2.6, there are two additional activities that have not been discussed before. The coloured figure is available on page 458. These are "Component (1,2...) Complete" and "Components Complete"; these are not additional

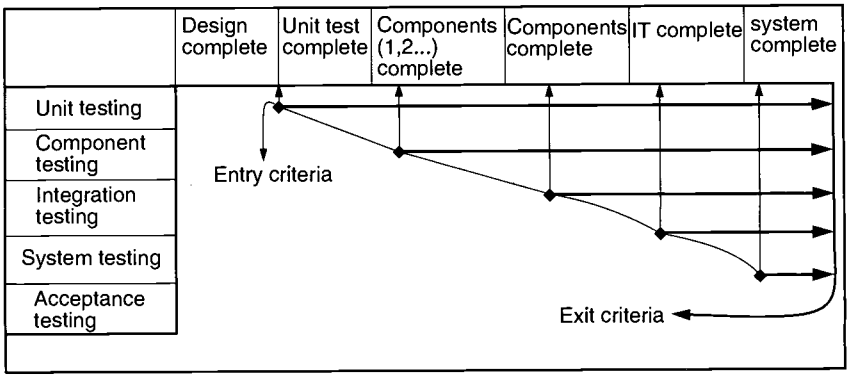


Figure 2.6
Modified V Model.

phases in a life cycle. They have been introduced just to denote that integration testing can start after two components have been completed, and when all components are integrated and tested, the next phase of testing, that is, system testing can start.

Table 2.3 Model applicability and relevance to verification and validation.

Models	Where Applicable	Relevant Verification and Validation (V & V) Issues
Waterfall	Where very clearly demarcated phases are present When deliverables of each phase can be frozen before proceeding to the next phase	Testing / V & V postponed by at least one phase Typically testing is among the most downstream activities Communication of error (and hence time for correction) can be high
Prototyping	Where we have a user (or a product manager) who can give feedback	Provides inbuilt feedback for the requirements Reuse of prototype (instead of throwing it away) can make verification and validation difficult and may produce undesirable effects
RAD	Where we have a user (or a product manager) who can give feedback When we have CASE and other modeling tools	Built-in feedback available beyond requirements also CASE tools can generate useful documentation that further enhances V & V
Spiral	Products evolving as increments Intermediate checking and correction is possible	Extends V & V to all increments Extends V & V to all phases (that is, those beyond requirements gathering as well) Enables the products to be demonstrated at any phase and enables frequent releases
V model	When design of tests can be separated from the actual execution	Early design of tests reduces overall delay by increasing parallelism between development and testing Early design of tests enables better and more timely validation of individual phases
Modified V model	When a product can be broken down into different parts, each of which evolves independently	Parallelism of V model further increased by making each part evolve independently Further reduces the overall delay by introducing parallelism between testing activities

2.5.6 Comparison of Various Life Cycle Models

As can be seen from the above discussion, each of the models has its advantages and disadvantages. Each of them has applicability in a specific scenario. Each of them also provides different issues, challenges, and opportunities for verification and validation. We summarize in Table 2.3 the salient points about applicability and relevance to verification and validation for each of the models.

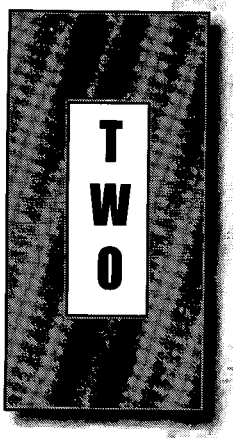
REFERENCES

The Waterfall Model was initially covered in [ROYC-70]. The origins of the Prototyping Model come from [BROO-75]. The Spiral Model was originally proposed in [BOEH-88]. [GRAD-97] provides some variations to the Spiral Model. [RAME-2002], [PRES-97] and [HUMP-86] provide overviews to all the models.

PROBLEMS AND EXERCISES

1. Which SDLC model would be most suitable for each of the following scenarios?
   a. The product is a bespoke product for a specific customer, who is always available to give feedback.
   b. The same as above, except that we also have access to a CASE tool that generates program code automatically.
   c. A general purpose product, but with a very strong product marketing team who understand and articulate the overall customer requirements very well.
   d. A product that is made of a number of features that become available sequentially and incrementally.
2. Which of the following products would you say is of "high quality," based on the criteria we discussed in the book? Justify your answer.
   a. Three successive versions of the product had respectively 0, 79, and 21 defects reported.
   b. Three successive versions of the product had respectively 85, 90, and 79 defects reported.
3. List three or more challenges from the testing perspective for each of the following models:
   a. Spiral Model.
   b. V Model.
   c. Modified V Model.

4. What are some of the challenges that you should expect when moving from the V Model to the Modified V Model?
5. Figure 2.1 gave the ETVX Diagram for the design phase of a software project. Draw a similar ETVX Diagram for the coding phase.
6. In the book we have discussed the Spiral Model as being ideally suited for a product that evolves in increments. Discuss how the V Model is applicable for such an incrementally evolving product.



Types of Testing

This part of the book discusses various types of tests. The chapters progress from the types of tests closer to code to those closer to users. White box testing, which tests the programs by having an internal knowledge of program code, is discussed in Chapter 3. Black box testing, which tests the product behavior by only knowing the external behavior as dictated by the requirements specifications, is discussed in Chapter 4. As software gets developed in a modular fashion and the modules have to be integrated together, integration testing is covered in Chapter 5. System and acceptance testing, which tests a product completely from a user's perspective in environments similar to customer deployments, is discussed in Chapter 6. Performance testing, which tests the ability of the system to withstand typical and excessive work loads, is discussed in Chapter 7. Since software is always characterized by change and since changes should not break what is working already, regression testing becomes very important and is discussed in Chapter 8. As software has to be deployed in multiple languages across the world, internationalization testing, the topic of Chapter 9, comes into play. Finally, adhoc testing, in Chapter 10, addresses the methods of testing a product in typical unpredictable ways that end users may subject the product to.

White Box Testing

CHAPTER 3

In this chapter—

- ✓ What is white box testing?
- ✓ Static testing
- ✓ Structural testing
- ✓ Challenges in white box testing

3.1 WHAT IS WHITE BOX TESTING?

Every software product is realized by means of a program code. White box testing is a way of testing the external functionality of the code by examining and testing the program code that realizes the external functionality. This is also known as *clear box*, or *glass box* or *open box* testing.

White box testing takes into account the program code, code structure, and internal design flow. In contrast, black box testing, to be discussed in Chapter 4, does not look at the program code but looks at the product from an external perspective.

A number of defects come about because of incorrect translation of requirements and design into program code. Some other defects are created by programming errors and programming language idiosyncrasies. The different methods of white box testing discussed in this chapter can help reduce the delay between the injection of a defect in the program code and its detection. Furthermore, since the program code represents what the product actually does (rather than what the product is intended to do), testing by looking at the program code makes us get closer to what the product is actually doing.

As shown in Figure 3.1, white box testing is classified into "static" and "structural" testing. The corresponding coloured version of Figure 3.1 is available on page 458. We will look into the details of static testing in Section 3.2 and take up structural testing in Section 3.3.

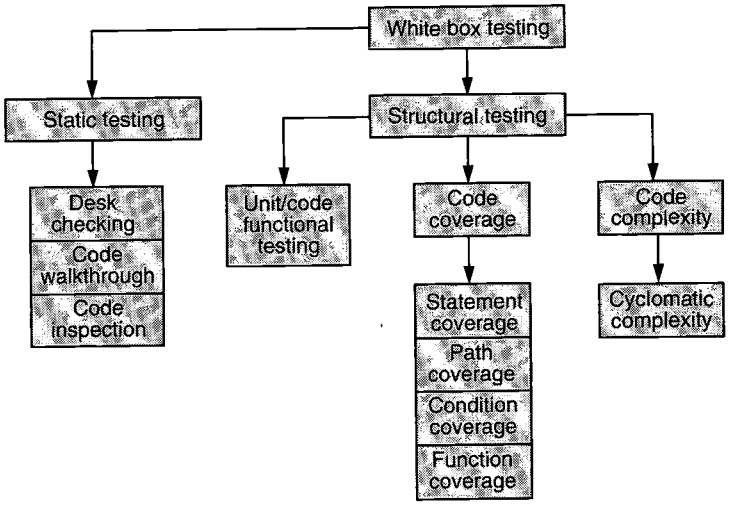


Figure 3.1
Classification of white box testing.

3.2 STATIC TESTING

Static testing is a type of testing which requires only the source code of the product, not the binaries or executables. Static testing does not involve

executing the programs on computers but involves select people going through the code to find out whether

- the code works according to the functional requirement;
- the code has been written in accordance with the design developed earlier in the project life cycle;
- the code for any functionality has been missed out ;
- the code handles errors properly.

Static testing can be done by humans or with the help of specialized tools.

3.2.1 Static Testing by Humans

These methods rely on the principle of humans reading the program code to detect errors rather than computers executing the code to find errors. This process has several advantages.

1. Sometimes humans can find errors that computers cannot. For example, when there are two variables with similar names and the programmer used a "wrong" variable by mistake in an expression, the computer will not detect the error but execute the statement and produce incorrect results, whereas a human being can spot such an error.
2. By making multiple humans read and evaluate the program, we can get multiple perspectives and therefore have more problems identified upfront than a computer could.
3. A human evaluation of the code can compare it against the specifications or design and thus ensure that it does what is intended to do. This may not always be possible when a computer runs a test.
4. A human evaluation can detect many problems at one go and can even try to identify the root causes of the problems. More often than not, multiple problems can get fixed by attending to the same root cause. Typically, in a reactive testing, a test uncovers one problem (or, at best, a few problems) at a time. Often, such testing only reveals the symptoms rather than the root causes. Thus, the overall time required to fix all the problems can be reduced substantially by a human evaluation.
5. By making humans test the code before execution, computer resources can be saved. Of course, this comes at the expense of human resources.
6. A proactive method of testing like static testing minimizes the delay in identification of the problems. As we have seen in Chapter 1, the sooner a defect is identified and corrected, lesser is the cost of fixing the defect.

7. From a psychological point of view, finding defects later in the cycle (for example, after the code is compiled and the system is being put together) creates immense pressure on programmers. They have to fix defects with less time to spare. With this kind of pressure, there are higher chances of other defects creeping in.

There are multiple methods to achieve static testing by humans. They are (in the increasing order of formalism) as follows.

1. Desk checking of the code
2. Code walkthrough
3. Code review
4. Code inspection

Since static testing by humans is done before the code is compiled and executed, some of these methods can be viewed as process-oriented or defect prevention-oriented or quality assurance-oriented activities rather than pure testing activities. Especially as the methods become increasingly formal (for example, Fagan Inspection), these traditionally fall under the "process" domain. They find a place in formal process models such as ISO 9001, CMMI, and so on and are seldom treated as part of the "testing" domain. Nevertheless, as mentioned earlier in this book, we take a holistic view of "testing" as anything that furthers the quality of a product. These methods have been included in this chapter because they have visibility into the program code.

We will now look into each of these methods in more detail.

3.2.1.1 Desk checking Normally done manually by the author of the code, desk checking is a method to verify the portions of the code for correctness. Such verification is done by comparing the code with the design or specifications to make sure that the code does what it is supposed to do and effectively. This is the desk checking that most programmers do before compiling and executing the code. Whenever errors are found, the author applies the corrections for errors on the spot. This method of catching and correcting errors is characterized by:

1. No structured method or formalism to ensure completeness and
2. No maintaining of a log or checklist.

In effect, this method relies completely on the author's thoroughness, diligence, and skills. There is no process or structure that guarantees or verifies the effectiveness of desk checking. This method is effective for correcting "obvious" coding errors but will not be effective in detecting errors that arise due to incorrect understanding of requirements or incomplete requirements. This is because developers (or, more precisely, programmers who are doing the desk checking) may not have the domain knowledge required to understand the requirements fully.

The main advantage offered by this method is that the programmer who knows the code and the programming language very well is well equipped to read and understand his or her own code. Also, since this is done by one individual, there are fewer scheduling and logistics overheads. Furthermore, the defects are detected and corrected with minimum time delay.

Some of the disadvantages of this method of testing are as follows.

1. A developer is not the best person to detect problems in his or her own code. He or she may be tunnel visioned and have blind spots to certain types of problems.
2. Developers generally prefer to write new code rather than do any form of testing! (We will see more details of this syndrome later in the section on challenges as well as when we discuss people issues in Chapter 13.)
3. This method is essentially person-dependent and informal and thus may not work consistently across all developers.

Owing to these disadvantages, the next two types of proactive methods are introduced. The basic principle of walkthroughs and formal inspections is to involve multiple people in the review process.

3.2.1.2 Code walkthrough This method and formal inspection (described in the next section) are group-oriented methods. Walkthroughs are less formal than inspections. The line drawn in formalism between walkthroughs and inspections is very thin and varies from organization to organization. The advantage that walkthrough has over desk checking is that it brings multiple perspectives. In walkthroughs, a set of people look at the program code and raise questions for the author. The author explains the logic of the code, and answers the questions. If the author is unable to answer some questions, he or she then takes those questions and finds their answers. Completeness is limited to the area where questions are raised by the team.

3.2.1.3 Formal inspection Code inspection—also called Fagan Inspection (named after the original formulator)—is a method, normally with a high degree of formalism. The focus of this method is to detect all faults, violations, and other side-effects. This method increases the number of defects detected by

1. demanding thorough preparation before an inspection/review;
2. enlisting multiple diverse views;
3. assigning specific roles to the multiple participants; and
4. going sequentially through the code in a structured manner.

A formal inspection should take place only when the author has made sure the code is ready for inspection by performing some basic desk checking and walkthroughs. When the code is in such a reasonable state of readiness,

an inspection meeting is arranged. There are four roles in inspection. First is the *author* of the code. Second is a *moderator* who is expected to formally run the inspection according to the process. Third are the *inspectors*. These are the people who actually provides, review comments for the code. There are typically multiple inspectors. Finally, there is a *scribe*, who takes detailed notes during the inspection meeting and circulates them to the inspection team after the meeting.

The author or the moderator selects the review team. The chosen members have the skill sets to uncover as many defects as possible. In an introductory meeting, the inspectors get copies (These can be hard copies or soft copies) of the code to be inspected along with other supporting documents such as the design document, requirements document, and any documentation of applicable standards. The author also presents his or her perspective of what the program is intended to do along with any specific issues that he or she may want the inspection team to put extra focus on. The moderator informs the team about the date, time, and venue of the inspection meeting. The inspectors get adequate time to go through the documents and program and ascertain their compliance to the requirements, design, and standards.

The inspection team assembles at the agreed time for the inspection meeting (also called the *defect logging meeting*). The moderator takes the team sequentially through the program code, asking each inspector if there are any defects in that part of the code. If any of the inspectors raises a defect, then the inspection team deliberates on the defect and, when agreed that there is a defect, classifies it in two dimensions—*minor/major* and *systemic/mis-execution*. A mis-execution defect is one which, as the name suggests, happens because of an error or slip on the part of the author. It is unlikely to be repeated later, either in this work product or in other work products. An example of this is using a wrong variable in a statement. Systemic defects, on the other hand, can require correction at a different level. For example, an error such as using some machine-specific idiosyncrasies may have to removed by changing the coding standards. Similarly, minor defects are defects that may not substantially affect a program, whereas major defects need immediate attention.

A scribe formally documents the defects found in the inspection meeting and the author takes care of fixing these defects. In case the defects are severe, the team may optionally call for a review meeting to inspect the fixes to ensure that they address the problems. In any case, defects found through inspection need to be tracked till completion and someone in the team has to verify that the problems have been fixed properly.

3.2.1.4 Combining various methods The methods discussed above are not mutually exclusive. They need to be used in a judicious combination to be effective in achieving the goal of finding defects early.

Formal inspections have been found very effective in catching defects early. Some of the challenges to watch out for in conducting formal inspections are as follows.

1. These are time consuming. Since the process calls for preparation as well as formal meetings, these can take time.
2. The logistics and scheduling can become an issue since multiple people are involved.
3. It is not always possible to go through every line of code, with several parameters and their combinations in mind to ensure the correctness of the logic, side-effects and appropriate error handling. It may also not be necessary to subject the entire code to formal inspection.

In order to overcome the above challenges, it is necessary to identify, during the planning stages, which parts of the code will be subject to formal inspections. Portions of code can be classified on the basis of their criticality or complexity as "high," "medium," and "low." High or medium complex critical code should be subject to formal inspections, while those classified as "low" can be subject to either walkthroughs or even desk checking.

Desk checking, walkthrough, review and inspection are not only used for code but can be used for all other deliverables in the project life cycle such as documents, binaries, and media.

3.2.2 Static Analysis Tools

The review and inspection mechanisms described above involve significant amount of manual work. There are several static analysis tools available in the market that can reduce the manual work and perform analysis of the code to find out errors such as those listed below.

1. whether there are unreachable codes (usage of `GOTO` statements sometimes creates this situation; there could be other reasons too)
2. variables declared but not used
3. mismatch in definition and assignment of values to variables
4. illegal or error prone typecasting of variables
5. use of non-portable or architecture-dependent programming constructs
6. memory allocated but not having corresponding statements for freeing them up memory
7. calculation of cyclomatic complexity (covered in the Section 3.3)

These static analysis tools can also be considered as an extension of compilers as they use the same concepts and implementation to locate errors. A good compiler is also a static analysis tool. For example, most C compilers provide different "levels" of code checking which will catch the various types of programming errors given above.

Some of the static analysis tools can also check compliance for coding standards as prescribed by standards such as POSIX. These tools can also check for consistency in coding guidelines (for example, naming conventions, allowed data types, permissible programming constructs, and so on).

While following any of the methods of human checking—desk checking, walkthroughs, or formal inspections—it is useful to have a code review checklist. Given below is checklist that covers some of the common issues. Every organization should develop its own code review checklist. The checklist should be kept current with new learning as they come about.

In a multi-product organization, the checklist may be at two levels—first, an organization-wide checklist that will include issues such as organizational coding standards, documentation standards, and so on; second, a product- or project-specific checklist that addresses issues specific to the product or project.

CODE REVIEW CHECKLIST

Data Item Declaration Related

- Are the names of the variables meaningful?
- If the programming language allows mixed case names, are there variable names with confusing use of lower case letters and capital letters?
- Are the variables initialized?
- Are there similar sounding names (especially words in singular and plural)? [These could be possible causes of unintended errors.]
- Are all the common structures, constants, and flags to be used defined in a header file rather than in each file separately?

Data Usage Related

- Are values of right data types being assigned to the variables?
- Is the access of data from any standard files, repositories, or databases done through publicly supported interfaces?
- If pointers are used, are they initialized properly?
- Are bounds to array subscripts and pointers properly checked?

- Has the usage of similar-looking operators (for example, = and == or & and && in C) checked?

Control Flow Related

- Are all the conditional paths reachable?
- Are all the individual conditions in a complex condition separately evaluated?
- If there is a nested IF statement, are the THEN and ELSE parts appropriately delimited?
- In the case of a multi-way branch like SWITCH / CASE statement, is a default clause provided? Are the breaks after each CASE appropriate?
- Is there any part of code that is unreachable?
- Are there any loops that will never execute?
- Are there any loops where the final condition will never be met and hence cause the program to go into an infinite loop?
- What is the level of nesting of the conditional statements? Can the code be simplified to reduce complexity?

Standards Related

- Does the code follow the coding conventions of the organization?
- Does the code follow any coding conventions that are platform specific (for example, GUI calls specific to Windows or Swing)

Style Related

- Are unhealthy programming constructs (for example, global variables in C, ALTER statement in COBOL) being used in the program?
- Is there usage of specific idiosyncrasies of a particular machine architecture or a given version of an underlying product (for example, using "undocumented" features)?

- Is sufficient attention being paid to readability issues like indentation of code?

MISCELLANEOUS

- Have you checked for memory leaks (for example, memory acquired but not explicitly freed)?

DOCUMENTATION RELATED

- Is the code adequately documented, especially where the logic is complex or the section of code is critical for product functioning?
- Is appropriate change history documented?
- Are the interfaces and the parameters thereof properly documented?

3.3 STRUCTURAL TESTING

Structural testing takes into account the code, code structure, internal design, and how they are coded. The fundamental difference between structural testing and static testing is that in structural testing tests are actually run by the computer on the built product, whereas in static testing, the product is tested by humans using just the source code and not the executables or binaries.

Structural testing entails running the actual product against some predesigned test cases to exercise as much of the code as possible or necessary. A given portion of the code is exercised if a test case causes the program to execute that portion of the code when running the test.

As discussed at the beginning of this chapter, structural testing can be further classified into unit/code functional testing, code coverage, and code complexity testing.

3.3.1 Unit/Code Functional Testing

This initial part of structural testing corresponds to some quick checks that a developer performs before subjecting the code to more extensive code coverage testing or code complexity testing. This can happen by several methods.

1. Initially, the developer can perform certain obvious tests, knowing the input variables and the corresponding expected output variables. This can be a quick test that checks out any obvious mistakes. By repeating these tests for multiple values of input variables, the confidence level of the developer to go to the next level

increases. This can even be done prior to formal reviews of static testing so that the review mechanism does not waste time catching obvious errors.

2. For modules with complex logic or conditions, the developer can build a "debug version" of the product by putting intermediate print statements and making sure the program is passing through the right loops and iterations the right number of times. It is important to remove the intermediate print statements after the defects are fixed.
3. Another approach to do the initial test is to run the product under a debugger or an Integrated Development Environment (IDE). These tools allow single stepping of instructions (allowing the developer to stop at the end of each instruction, view or modify the contents of variables, and so on), setting break points at any function or instruction, and viewing the various system parameters or program variable values.

All the above fall more under the "debugging" category of activities than under the "testing" category of activities. All the same, these are intimately related to the knowledge of code structure and hence we have included these under the "white box testing" head. This is consistent with our view that testing encompasses whatever it takes to detect and correct defects in a product.

3.3.2 Code Coverage Testing

Since a product is realized in terms of program code, if we can run test cases to exercise the different parts of the code, then that part of the product realized by the code gets tested. Code coverage testing involves designing and executing test cases and finding out the percentage of code that is covered by testing. The percentage of code covered by a test is found by adopting a technique called *instrumentation* of code. There are specialized tools available to achieve instrumentation. Instrumentation rebuilds the product, linking the product with a set of libraries provided by the tool vendors. This instrumented code can monitor and keep an audit of what portions of code are covered. The tools also allow reporting on the portions of the code that are covered frequently, so that the critical or most-often portions of code can be identified.

Code coverage testing is made up of the following types of coverage.

1. Statement coverage
2. Path coverage
3. Condition coverage
4. Function coverage

3.3.2.1 Statement coverage Program constructs in most conventional programming languages can be classified as

1. Sequential control flow
2. Two-way decision statements like `if then else`
3. Multi-way decision statements like `Switch`
4. Loops like `while do`, `repeat until` and `for`

Object-oriented languages have all of the above and, in addition, a number of other constructs and concepts. We will take up issues pertaining to object oriented languages together in Chapter 11. We will confine our discussions here to conventional languages.

Statement coverage refers to writing test cases that execute each of the program statements. One can start with the assumption that more the code covered, the better is the testing of the functionality, as the code realizes the functionality. Based on this assumption, code coverage can be achieved by providing coverage to each of the above types of statements.

For a section of code that consists of statements that are sequentially executed (that is, with no conditional branches), test cases can be designed to run through from top to bottom. A test case that starts at the top would generally have to go through the full section till the bottom of the section. However, this may not always be true. First, if there are asynchronous exceptions that the code encounters (for example, a divide by zero), then, even if we start a test case at the beginning of a section, the test case may not cover all the statements in that section. Thus, even in the case of sequential statements, coverage for all statements may not be achieved. Second, a section of code may be entered from multiple points. Even though this points to not following structured programming guidelines, it is a common scenario in some of the earlier programming languages.

When we consider a two-way decision construct like the `if` statement, then to cover all the statements, we should also cover the `then` and `else` parts of the `if` statement. This means we should have, for each `if then else`, (at least) one test case to test the `Then` part and (at least) one test case to test the `else` part.

The multi-way decision construct such as a `Switch` statement can be reduced to multiple two-way `if` statements. Thus, to cover all possible `switch` cases, there would be multiple test cases. (We leave it as an exercise for the reader to develop this further.)

Loop constructs present more variations to take care of. A loop—in various forms such as `for`, `while`, `repeat`, and so on—is characterized by executing a set of statements repeatedly until or while certain conditions are met. A good percentage of the defects in programs come about because of loops that do not function properly. More often, loops fail in what are called "boundary conditions." One of the common looping errors is that the termination condition of the loop is not properly stated. In order to make

sure that there is better statement coverage for statements within a loop, there should be test cases that

1. Skip the loop completely, so that the situation of the termination condition being true before starting the loop is tested.
2. Exercise the loop between once and the maximum number of times, to check all possible "normal" operations of the loop.
3. Try covering the loop, around the "boundary" of **n**—that is, just below **n**, **n**, and just above **n**.

Statement Coverage = (Total statements exercised / Total number of executable statements in program) * 100

The statement coverage for a program, which is an indication of the percentage of statements actually executed in a set of tests, can be calculated by the formula given alongside in the margin.

It is clear from the above discussion that as the type of statement progresses from a simple sequential statement to `if then else` and through to loops, the number of test cases required to achieve statement coverage increases. Taking a cue from the Dijkstra's Doctrine in Chapter 1, just as exhaustive testing of all possible input data on a program is not possible, so also exhaustive coverage of all statements in a program will also be impossible for all practical purposes.

Even if we were to achieve a very high level of statement coverage, it does not mean that the program is defect-free. First, consider a hypothetical case when we achieved 100 percent code coverage. If the program implements wrong requirements and this wrongly implemented code is "fully tested," with 100 percent code coverage, it still is a wrong program and hence the 100 percent code coverage does not mean anything.

Next, consider the following program.

```
Total = 0; /* set total to zero */
if (code == "M") {
     stmt1;
     stmt2;
     Stmt3;
     stmt4;
     Stmt5;
     stmt6;
     Stmt7;
}

else percent = value/Total*100; /* divide by zero */
```

In the above program, when we test with code = "M," we will get 80 percent code coverage. But if the data distribution in the real world is such that 90 percent of the time, the value of code is not = "M," then, the program will fail 90 percent of the time (because of the divide by zero in the highlighted line). Thus, even with a code coverage of 80 percent, we are left with a defect that hits the users 90 percent of the time. Path coverage, discussed in Section 3.3.2.2, overcomes this problem.

Path Coverage = (Total paths exercised/ Total number of paths in program) * 100

3.3.2.2 Path coverage In path coverage, we split a program into a number of distinct paths. A program (or a part of a program) can start from the beginning and take any of the paths to its completion.

Let us take an example of a date validation routine. The date is accepted as three fields `mm`, `dd` and `yyyy`. We have assumed that prior to entering this routine, the values are checked to be numeric. To simplify the discussion, we have assumed the existence of a function called `leapyear` which will return `TRUE` if the given year is a leap year. There is an array called `DayofMonth` which contains the number of days in each month. A simplified flow chart for this is given in Figure 3.2 below.

As can be seen from the figure, there are different paths that can be taken through the program. Each part of the path is shown in red. The coloured representation of Figure 3.2 is available on page 459. Some of the paths are

- A
- B-D-G
- B-D-H
- B-C-E-G
- B-C-E-H
- B-C-F-G
- B-C-F-H

Regardless of the number of statements in each of these paths, if we can execute these paths, then we would have covered most of the typical scenarios.

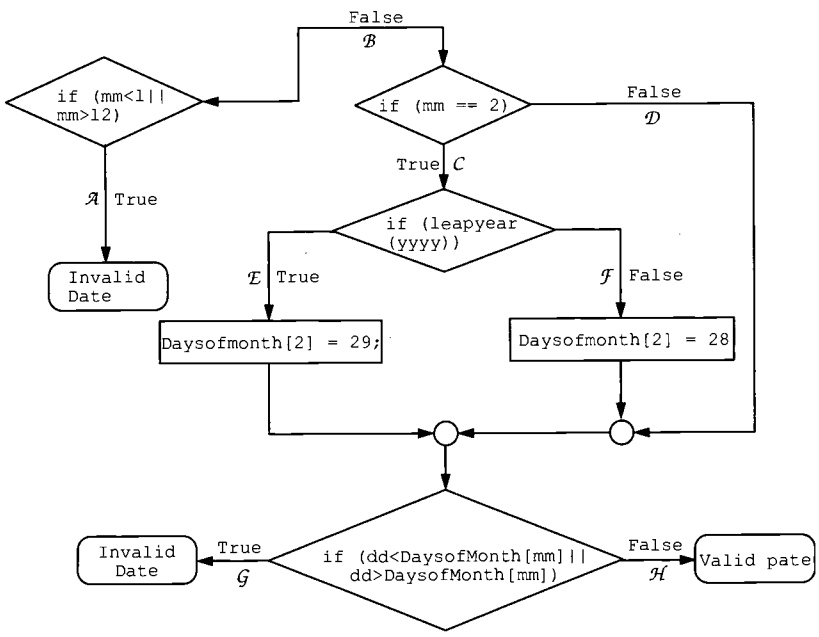


Figure 3.2
Flow chart for a date validation routine.